Constraints Guide

UG625 (v 11.4) December 2, 2009

XILINX®

Xilinx Trademarks and Copyright Information

Table of Contents

XILINX®

Chapter 1

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

This Guide contains the following chapters:

- About this Guide, discusses what is new in this Guide for the ISE® Design Suite and provides other introductory information.
- 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, routing directives, synthesis constraints, timing constraints
- 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.
- Timing Constraint Strategies, provides general guidelines that explain how to constrain the timing on designs when using the implementation tools for FPGA devices.
- 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.

What's New in this Release

This section shows what is new in this guide.

- Added the following new constraints:
  - DEFAULT
  - MAX_FANOUT
  - IODELAY_GROUP
  - MIODELAY_GROUP

- Removed all deprecated architectures. See Constraints by Architecture
- Added ICAP to the list of LOC constraint targets.
- Added CLOCK REGION description to the AREA_GROUP constraint.
- Removed SET command.
- Added lat_ce_q to ENABLE and DISABLE constraints.

- TNM constraint is no longer supported for schematic designs.
- FORCE value added for IOB constraint.
- TEMPERATURE constraint updated with support for Spartan-3A DSP architectures.
- Added information about PlanAhead™ software.
- PACE is supported for CPLDs not FPGAs.
- Removed USELOWSKEWLINES constraint.

Additional Resources

To find additional documentation, see the Xilinx website at:

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

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

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

Conventions

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

Typographical

The following typographical conventions are used in this document:

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

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

Convention	Meaning or Use	Example
		.
Horizontal ellipsis . . .	Repetitive material that has been omitted	**allow block** . . . *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link	See the section Additional Resources for details. Refer to Title Formats in Chapter 1 for details. See Figure 2-5 in the Virtex®-6 Handbook.

Chapter 2

Architecture Support

This chapter provides information on constraints supported by each architecture. See Constraints by Architectures

Constraints by Architectures

The following architectures are discussed in this Guide.

- CoolRunner™ XPLA3, and CoolRunner-II, XC9500, and XC9500XL CPLDs
- Spartan®-3, Spartan-3A, Spartan-3E, Spartan-3L, Spartan-6, Virtex®-4, Virtex-5, and Virtex-6.

Chapter 3

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 components 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.

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

The *Xilinx Constraints Guide* documents Implementation constraints.

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 for Timing

For a timing specification (TIMESPEC), 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 TNM (Timing Name) (pronounced tee-name) and TNM_NET (Timing Name Net) constraints.
- Create groups that are combinations of existing groups using TIMEGRP (Timing Group) symbols.
- Create groups by pattern matching on net names. For more information, see Creating Groups by Pattern Matching in the TIMEGRP (Timing Group) 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.

Predefined Groups Keyword

Keyword	Description
CPUS	PPC405 in Virtex®-4 FX devices
DSPS	DSP48 and any DSP48 derivative in Virtex-4, Virtex-5, and Spartan®-3A Extended devices CPUS in Virtex-5 FXT devices
FFS	All CLB and IOB edge-triggered flip-flops and Shift Register LUTs (all devices have shift register LUTs)
HSIOS	GT11 (Virtex-4), GTP_DUAL (Virtex-5) and GTX_DUAL (Virtex-5 FXT)
LATCHES	All CLB and IOB level-sensitive latches
MULTS	Synchronous and asynchronous multipliers.
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) – FIFOS – All FIFO (First In, First Out) Block RAM Memory
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 constraints 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.

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)

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 tool)
- RAM_STYLE (XST tool)

Synthesis constraints are documented as follows:

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

The following constraints are synthesis constraints:

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

Timing Constraints

The Xilinx® software enables you to specify precise timing requirements for your Xilinx designs. These requirements can be specified using global or path specific timing constraints. The recommended method for defining the constraints for common timing requirement scenarios is discussed further in Chapter 5 of this manual.

The primary method of specifying timing constraints is by entering them in the User Constraints File (UCF). In addition, constraints can be entered in the source of the design (HDL and Schematic). For detailed information on each constraint, see the later chapters of 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.

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. See the Xilinx *XST User Guide* for more information.

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 "TS` *name*`"=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 output 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

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/PLL/MMCMs in all FPGA devices

For more information, see PERIOD Specifications on CLKDLLs, DCMs, PLLs, and MMCMs 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. 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 4

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.

Constraints Entry Methods

ISE Tool	Constraint Type	Devices
Constraints Editor	Timing	All CPLD and FPGA device families
PlanAhead™	IO placement and area-group constraints	All FPGA device families
PACE	IO placement	All CPLDs device families
Schematic and Symbol Editors	IO placement and other placement 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.

Constraints Entry Table

Constraint	Schematic	VHDL Verilog	NCF	UCF	Constraints Editor	PCF	XCF	PlanAhead	PACE	FPGA Editor	Project Navigator
AREA_GROUP	Yes		Yes	Yes	Yes			Yes			
ASYNC_REG		Yes	Yes	Yes	Yes						
BEL		Yes	Yes	Yes				Yes			
BLKNM	Yes	Yes	Yes	Yes			Yes				
BUFG (CPLD)	Yes	Yes	Yes	Yes			Yes				
CLOCK_DEDICATED_ROUTE			Yes	Yes							
COLLAPSE	Yes	Yes	Yes	Yes							
COMPGRP						Yes					
CONFIG_MODE				Yes							
COOL_CLK	Yes	Yes	Yes	Yes							
DATA_GATE	Yes	Yes	Yes	Yes							
DEFAULT	Yes	Yes	Yes	Yes			Yes	Yes	Yes		
DCI_CASCADE			Yes	Yes		Yes					
DCI_VALUE			Yes	Yes							
DIRECTED_ROUTING			Yes	Yes						Yes	
DISABLE			Yes	Yes		Yes					
DRIVE	Yes	Yes	Yes	Yes			Yes	Yes	Yes	Yes	
DROP_SPEC	Yes		Yes	Yes		Yes					
ENABLE			Yes	Yes		Yes					
ENABLE_SUSPEND			Yes	Yes							
FAST	Yes	Yes	Yes	Yes			Yes	Yes	Yes		
FEEDBACK				Yes	Yes	Yes	Yes	Yes			
FILE	Yes	Yes									
FLOAT	Yes	Yes	Yes	Yes			Yes				
FROM-THRU-TO			Yes	Yes	Yes	Yes		Yes			
FROM-TO			Yes	Yes	Yes	Yes	Yes	Yes			
HBLKNM	Yes	Yes	Yes	Yes							
HLUTNM	Yes	Yes	Yes	Yes	Yes		Yes				
HU_SET	Yes	Yes	Yes	Yes			Yes				
IBUF_DELAY_VALUE	Yes	Yes	Yes	Yes							
IFD_DELAY_VALUE	Yes	Yes	Yes	Yes							
INREG	Yes			Yes							
IOB	Yes	Yes	Yes	Yes			Yes				Yes
IOBDELAY	Yes	Yes	Yes	Yes	Yes						
IODELAY_GROUP				Yes							

Constraint	Schematic	VHDL Verilog	NCF	UCF	Constraints Editor	PCF	XCF	PlanAhead	PACE	FPGA Editor	Project Navigator
IOSTANDARD	Yes	Yes	Yes	Yes			Yes	Yes	Yes	Yes	
KEEP	Yes	Yes	Yes	Yes			Yes				
KEEPER	Yes	Yes	Yes	Yes	Yes		Yes			Yes	
KEEP_HIERARCHY	Yes	Yes	Yes	Yes			Yes				Yes
LOC	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes		
LOCATE						Yes				Yes	
LOCK_PINS		Yes	Yes	Yes							
LUTNM	Yes	Yes	Yes	Yes							
MAP	Yes		Yes	Yes							
MAXDELAY	Yes	Yes	Yes	Yes	Yes	Yes				Yes	
MAX_FANOUT		Yes					Yes				Yes
MAXPT		Yes	Yes	Yes							
MAXSKEW	Yes	Yes	Yes	Yes	Yes	Yes				Yes	
NODELAY	Yes	Yes	Yes	Yes			Yes				
IODELAY_GROUP				Yes							
NOREDUCE	Yes	Yes	Yes	Yes			Yes				
OFFSET IN	Yes		Yes	Yes	Yes	Yes	Yes	Yes			
OFFSET OUT	Yes		Yes	Yes	Yes	Yes	Yes	Yes			
OPEN_DRAIN	Yes	Yes	Yes	Yes			Yes				
OPT_EFFORT	Yes		Yes	Yes							Yes
OPTIMIZE	Yes	Yes	Yes	Yes							Yes
PERIOD	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes		Yes	
PIN				Yes							
POST_CRC				Yes		Yes					
POST_CRC_ACTION				Yes		Yes					
POST_CRC_FREQ				Yes		Yes					
POST_CRC_SIGNAL				Yes		Yes					
PRIORITY			Yes	Yes		Yes					
PROHIBIT				Yes		Yes		Yes	Yes	Yes	
PULLDOWN	Yes	Yes	Yes	Yes			Yes	Yes	Yes	Yes	
PULLUP	Yes	Yes	Yes	Yes			Yes	Yes	Yes	Yes	
PWR_MODE	Yes	Yes	Yes	Yes			Yes				
REG	Yes	Yes	Yes	Yes			Yes				
RLOC	Yes	Yes	Yes	Yes			Yes	Yes			
RLOC_ORIGIN	Yes	Yes	Yes	Yes		Yes		Yes			
RLOC_RANGE	Yes	Yes	Yes	Yes		Yes	Yes				
SAVE NET FLAG	Yes	Yes	Yes	Yes			Yes				

Constraint	Schematic	VHDL Verilog	NCF	UCF	Constraints Editor	PCF	XCF	PlanAhead	PACE	FPGA Editor	Project Navigator
SCHMITT_TRIGGER	Yes	Yes	Yes	Yes			Yes				
SLEW	Yes	Yes	Yes	Yes			Yes	Yes	Yes	Yes	
SLOW	Yes	Yes	Yes	Yes			Yes	Yes	Yes	Yes	
STEPPING				Yes							
SUSPEND	Yes	Yes	Yes	Yes					Yes		
SYSTEM_JITTER	Yes	Yes	Yes	Yes			Yes				
TEMPERATURE			Yes	Yes	Yes	Yes					
TIG	Yes		Yes	Yes	Yes	Yes	Yes	Yes			
TIMEGRP			Yes	Yes	Yes	Yes	Yes	Yes			
TIMESPEC			Yes	Yes	Yes		Yes	Yes			
TNM			Yes	Yes	Yes		Yes	Yes			
TNM_NET	Yes		Yes	Yes	Yes		Yes	Yes			
TPSYNC	Yes		Yes	Yes							
TPTHRU	Yes		Yes	Yes	Yes						
TSidentifier			Yes	Yes	Yes	Yes	Yes			Yes	
U_SET	Yes	Yes	Yes	Yes			Yes				
USE_RLOC	Yes	Yes	Yes	Yes			Yes	Yes			
VCCAUX			Yes	Yes							
VOLTAGE			Yes	Yes	Yes	Yes					
VREF	Yes		Yes	Yes							
WIREAND	Yes	Yes	Yes	Yes							
XBLKNM	Yes	Yes	Yes	Yes			Yes				

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.

Example 1:

```
attribute RLOC : string;
attribute RLOC of u123 :  label is "R11C1.S0";
```

Example 2:

```
attribute bufg:  string;
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.

Note The signal attribute must be used on the output port.

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

Attributes are bounded by asterisks (*), and use the following syntax:

(* *attribute_name* = *attribute_value* ***)**

where

- The *attribute* precedes the signal, module, or instance declaration to which it refers.
- The *attribute_value* is a string. No integer or scalar values are allowed.
- The *attribute_value* is between quotes.
- The default is 1. **(* attribute_name *)** is the same as **(* attribute_name = "1" *)**.

Verilog Attributes Syntax Example One

```
(* clock_buffer = "IBUFG" *) input CLK;
```

Verilog Attributes Syntax Example Two

```
(* INIT = "0000" *) reg [3:0] d_out;
```

Verilog Attributes Syntax Example Three

```
always@(current_state or reset)
  begin (* parallel_case *) (* full_case *)
   case (current_state)
```

Verilog Attributes Syntax Example Four

```
(* mult_style = "pipe_lut" *) MULT my_mult (a, b, c);
```

Verilog Limitations

Verilog attributes are not supported for:

- Signal declarations
- Statements
- Port connections
- Expression operators

Verilog Meta Comments

Constraints can also be specified in Verilog code using meta comments. The Verilog format is the preferred syntax, but the meta comment style is still supported. Use the following syntax:

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

Examples

```
// synthesis attribute RLOC of u123 is R11C1.S0
// synthesis attribute HU_SET u1 MY_SET
// synthesis attribute bufg of my_clock is "clk"
```

UCF

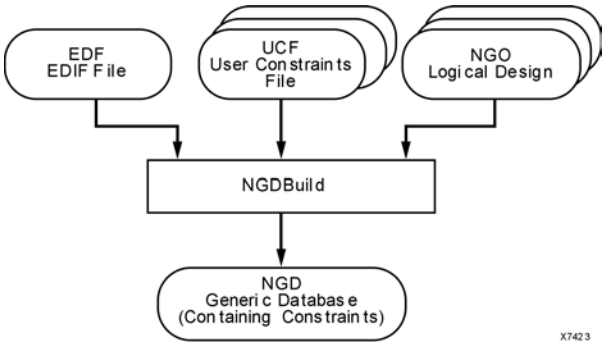
UCF files are ASCII files specifying constraints on the logical design. You can create these files and enter your constraints with any text editor. You can also use the Constraints Editor to create constraints within UCF files. For more information, see Constraints Editor in this chapter.

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

UCF Flow

The following figure illustrates the UCF flow.

UCF Files Flow



UCF files are input to NGDBuild (see the preceding figure). The constraints in the UCF files 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

In addition to entering timing constraints through Constraints Editor, you can manually enter timing specifications as constraints in UCF files. When you run NGDBuild on the design, the timing constraints are added to the design database as part of the NGD file. You can also use the Xilinx® Constraints Editor to enter timing constraints in UCF files.

Constraint Conflicts in Multiple UCF Files

The Xilinx software still uses "last constraint wins" just like for HDL/NCF/UCF/PCF processing. Currently, the UCF files are processed in the order in which they are added to the project (either in the Project Navigator or via Tcl command), and it has no bearing on timestamps or the order in which the files were modified.

UCF and NCF File Syntax

Logical constraints are found in:

- The Netlist Constraint File (NCF), an ASCII file typically 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 or NCF files and not in the PCF file.

General Rules for UCF and NCF

- 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 below.

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, HDL, UCF or NCF files.. If the constraints overlap, UCF overrides NCF and schematic/netlist constraints. NCF overrides schematic/netlist constraints.

If by mistake two or more elements are locked onto a single location, MAP 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*;

- *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.

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 SLICE in column 8.

```
INST "/loads_of_logic/*" LOC=SLICE_X*Y8;
```

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

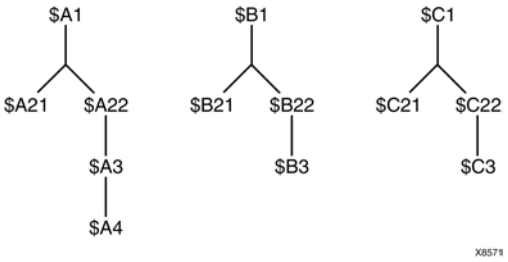
Top-level block names (design names) are ignored when searching for instance name matches. You can use the asterisk wildcard character (*) to traverse the hierarchy of a design within a UCF and NCF file. The following syntax applies (where level1 is an example hierarchy level name).

UCF Design Hierarchy

*	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.

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 or an IOB.
- 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 semicolon.

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

The Constraints Editor is a graphical tool that simplifies the process of entering timing constraints. This tool guides you through the process of creating constraints without requiring you to understand UCF 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 (NGCBuild), 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, NGCBuild 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:

- 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. NGCBuild (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.

Note Not all Xilinx® constraints are accessible through Constraints Editor.

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*

ngdfile_name is the name of the NGD file

It is 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*

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

It is necessary to use the .ucf extension.

As a Background Process

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

```
constraints_editor &
```

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.

PlanAhead

You can use PlanAhead™ software either before or after synthesis. PlanAhead software is invoked for all Virtex®-4 and higher and Spartan®-3 and higher devices. Using a drag-and-drop approach, you can enter placement constraints, including pinout constraints, logic placement, and area constraints. For details and device support information, see *PlanAhead User Guide*.

Assigning Placement Constraints

When targeting FPGAs, you can use PlanAhead software to enter placement constraints that control I/O pin and logic assignments, global logic placement, and area group assignment. PlanAhead software is now automatically invoked at various stages of the design process to allow you to analyze the design and to apply placement constraints. A simplified version of PlanAhead software is invoked from Project Navigator to enable only the types of features required to perform the selected tasks. Standalone PlanAhead has many more features available.

You should be aware of certain behaviors when PlanAhead software is invoked from Project Navigator. It is a separate CPU process and is not communicating "realtime" with Project Navigator as some of the other tools do. You should avoid making updates to Project Navigator source files while PlanAhead is invoked to prevent data mismatch or out of sync issues.

When PlanAhead is invoked, the appropriate source files are passed to PlanAhead to populate the PlanAhead Project. When the PlanAhead project is saved, only the modified UCF files are passed back to Project Navigator to update the project. These input source files vary depending on the process step invoked.

For more information on what types of files are passed, refer to the *Pin Assignment* and *Floorplanning and Placement Constraints* sections later in this chapter. The following sections cover strategies for entering placement constraints using PlanAhead.

Defining I/O Pin Configurations

Pin Assignment Overview

PinAhead can be invoked as a standalone tool or from within Project Navigator. Invoking PinAhead standalone can be helpful early in the design process when HDL sources may not yet be complete. I/O ports can be defined manually within the tool or by importing a CSV format spreadsheet or HDL sources. You can define an initial pinout and export a UCF file for use in the Project Navigator flow.

A UCF file is required when invoking PinAhead from within Project Navigator. If a UCF does not exist, an empty one is created. Creation of I/O ports manually or by importing a CSV spreadsheet is not enabled when invoking PinAhead from Project Navigator.

PinAhead is an I/O pin assignment environment containing many helpful views and capabilities. You can selectively drag and drop groups of I/O ports onto the device using a variety of methods. An automatic placement routine is also available. Comprehensive Design Rule Checks (DRCs) ensure legal pinout definition.

Reviewing I/O Pin Data Information

Data Sheets provide device specifications, including I/O standards. To get device-specific I/O standard information, see the data sheet for the device you are targeting. A lot of the data contained in the data sheets is also available inside of the PinAhead tool. The types of information available include I/O standards, clock capable pins, internal trace delays, differential pairs, clock region and I/O bank contents, etc. Information about I/O related device resources such as global and regional clock buffers, I/O delays and delay controllers, gigabit transceivers, etc. is also available.

Pin Assignment

To invoke PinAhead standalone either click the PlanAhead Windows Desktop icon or enter planAhead on the Linux command line. From Project Navigator, you can use any of the following methods to start your pin assignment process, which allows you to choose the method most convenient for you:

- Floorplanning I/O – Pre-Synthesis

  When using this command or process step, the HDL source files are passed to PlanAhead in order to extract the top level I/O port information only. If a UCF files exists in the Project Navigator project, it is passed to PlanAhead for modification. If a UCF does not exist, you are prompted to create one. If multiple UCF files exist, you are prompted to select the desired file to add new constraints to. Existing constraints are modified in whichever file they are contained in.

  Refer to the PinAhead Documentation section for information on using the PinAhead environment contained in PlanAhead.

  Once the I/O pin assignment is made, you save the PlanAhead project and exit PlanAhead. This updates the UCF files in the Project Navigator project and update the project status accordingly. Exiting PlanAhead without saving will not change the Project Navigator UCF source files or status.

- Floorplanning a Design – Post-Synthesis

  When using this command or process step, the synthesized netlist source files are passed to PlanAhead. If a UCF files exists in the Project Navigator project, it is passed to PlanAhead for modification. If a UCF does not exist, you are prompted to create one. If multiple UCF files exist, you are prompted to select the desired file to add new constraints to. Existing constraints are modified in whichever file they are contained in.

  Having a synthesized netlist as input enables more functionality in PinAhead since the tool is now aware of the clocks and clock related logic in the design. Additional I/O planning capabilities and DRCs are provided to make more intelligent pin assignment decisions. The design connectivity can also be analyzed to ensure optimized use of device resources in relation to the I/Os.

  Refer to the PinAhead Documentation section for information on using the PinAhead environment contained in PlanAhead.

  Once the I/O pin assignment is made, you will then save the PlanAhead project and exit PlanAhead. This will update the UCF files in the Project Navigator project and update the project status accordingly. Exiting PlanAhead without saving with not change the Project Navigator UCF source files or status.

PinAhead Documentation

The *PlanAhead User Guide* contains a section on *I/O Pin Planning with PinAhead* that provides information on using PinAhead for analyzing the device resources and I/O pin assignment.

The *PlanAhead Tutorial* contains a section on *I/O Pin Assignment with PinAhead* for to help you quickly get up to speed with PinAhead.

A PinAhead video demonstration is also available at www.xilinx.com/design.

Floorplanning and Placement Constraints

PlanAhead provides a comprehensive environment for analyzing the design from a number of different aspects including connectivity, density and timing. You can then apply placement constraints to help drive the implementation tools toward better or more consistent results. These constraints may include LOC constraints to lock specific logic objects into specific sites on the device or AREA_GROUP constraints to constrain a group of logic within a specific area of the device.

For more information on the types of placement constraints available, see Constraints Overview.

Placement LOC Constraint Assignment

PlanAhead enables you to lock down any logic to specific device sites. This often includes global logic objects such as the following: BUFG, BRAM, MULT, PPC405, GT, DLL, and DCM.

You can place logic objects by simply dragging the desired logic object from any of the appropriate PlanAhead views and drop it in the Device View in the Workspace. Some types of logic such as I/O ports enable you to enter the desired location site in the object General Properties view.

For more information about assigning placement constraints, see the Floorplanning the Design -> Using Placement Constraints section of the *PlanAhead Users Guide*.

Area Group Assignment

Area groups are the primary means of placing logic in specific regions of the device, for example, within a particular clock region. PlanAhead enables you to create area groups using a wide variety of methods. Assistance with connectivity, size logic types and ranges are all provided by the tool including DRCs to ensure proper AREA_GROUP property definition.

For more information about creating area group constraints, see the Floorplanning the Design section of the *PlanAhead Users Guide*.

PACE

For CPLDs, you can set constraints in PACE (Pinout & Area Constraints Editor). 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. For more information about accessing and using PACE , see the ISE® Help.

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 PlanAhead™ or PACE, see the ISE® Help. PACE is supported for CPLDs PlanAhead is supported for FPGAs.

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 PlanAhead or 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. PlanAhead and 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.

Constraints Supported in FPGA Editor Constraint

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.

XCF

XST constraints can be specified in theXilinx® Constraint File (XCF). The XCF has an extension of .xcf. For information on specifying the XCF in the ISE® Design Suite, see the ISE Help. Also see the XST Constraint File section in the *XST User Guide*.

Constraints Priority

In some cases more than one timing constraint covers the same path in the design. In these cases, that constraint conflict must be resolved with the higher priority constraint taking precedence and being applied to the path, and the lower priority constraints being ignored for that path. The method of constraints resolution depends on both the order of constraint specification as well as the priority of the constraints specified. The rules of constraint priority resolution are described below. This determination is based upon the constraint prioritization or which constraint appears later in the PCF file, if there are overlapping constraints of the same priority. For example, if the design has two PERIOD constraints that cover the same paths, the later PERIOD constraint in the PCF file covers or analyzes these paths. The previous PERIOD constraints have “0 items analyzed” in the timing report. In order to modify the default constraint resolution behavior, the constraint priority can be assigned using the PRIORITY keyword.

File Priorities

When conflicting constraints have the same priority, the order of specification is used to determine the constraint that takes precedence. The resolution rule for identical priority constraints is the constraint that is specified last overwrites any previously defined constraints. This rule applies to constraints within a single UCF file as well as constraints defined in multiple UCF files.

The following list defines the precedence order of identical priority constraints when these constraints are defined in different constraint files. The list is given in descending priority order with the highest priority constraint listed first.

- 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 or Constraints specified in HDL that are passed down in the netlist

Timing Specification Priorities

In cases where two different constraints cover the same path in the design the constraint with the highest priority takes precedence and be applied to that path while other is ignored. The rules for defining the priority of different constraints is given in the list below. The list is defined in descending priority order with the highest priority constraint listed first.

- Timing Ignore (TIG)
- FROM THRU TO
  - Source and Destination are User Defined Groups
  - Source or Destination are User Defined Groups
  - Source and Destination are Pre-defined Groups
- FROM TO
  - Source and Destination are User Defined Groups
  - Source or Destination are User Defined Groups
  - Source and Destination are Pre-defined Groups
- OFFSET Constraints
  - Specific Data IOB (NET OFFSET) Constraint
  - Time Group of Data IOBs (Grouped OFFSET) Constraint
  - All Data IOBs (Global OFFSET) Constraint
- PERIOD

OFFSET Priorities

If two specific OFFSET constraints at the same level of precedence cover the same path, 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.

MAXSKEW and MAXDELAY Priorites

Net delay and net skew specifications are analyzed independently of path delay analysis and do not interfere with one another. NET TIG do interact with the NET constraints and take precedence.

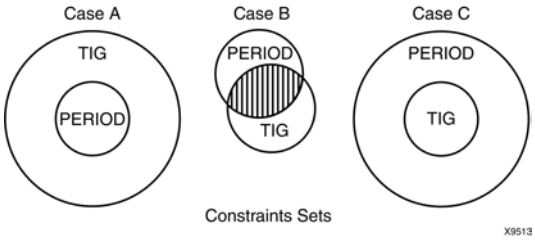
Constraints Priority Exceptions

The PRIORITY keyword can be used to over-ride the default constraint resolution behavior. The PRIORITY keyword uses a value of —255 to +255 to manually assign a priority to a constraint. The smaller the constraint value, the higher the constraint priority. This keyword only affects constraint resolution priority, and does not influence the priority in which the implementation tools place and route the resources covered by the constraint. A detailed description of the priority constraint is presented in Chapter 6 of this document.

Constraint Set Interaction

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.

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.

Chapter 5

Timing Constraint Strategies

The goal of using timing constraints is to ensure that all the design requirements are communicated to the implementation tools. This goal also implies that all paths in the design are covered by the appropriate constraint. This chapter provides general guidelines that explain the strategy for identifying and constraining the most common timing paths in FPGA devices in the most efficient manner possible. This chapter contains the following sections:

Basic Constraints Methodology

In order to ensure proper operation of the design, the timing requirements for all paths in the design must be communicated to the implementation tools. The timing requirements of a design can be broken down into several global categories based on the type of path that is to be covered. The most common types of path categories include: input paths, register-to-register paths, output paths, and path specific exceptions. Associated with each of these global category types is a Xilinx® timing constraint. The most efficient method for specifying these constraints is to begin with global constraints and add path specific exceptions as needed. In many cases, only the global constraints will be required.

The FPGA implementation tools are driven by the specified timing requirements and will assign device resources and expend the appropriate amount of effort necessary to ensure the timing requirements are met. However, when a requirement is over-constrained – or specified as a value greater than the design requirement - the effort spent by the tools to meet this constraint increases significantly. This extra effort results in increased memory use and tool runtime. Also, more importantly, over-constraint can result in loss of performance for not only the constraint in question, but in other constraints as well. For this reason, Xilinx recommends that the constraint values be specified using the actual design requirements.

The method of applying constraints given in this guide uses UCF constraint syntax examples. This format is used to highlight the constraints syntax that conveys the design requirements. However, the easiest method of entering design constraints is to use the Constraints Editor tool. This tool provides a unified location in which to manage all of the timing constraints associated with a design as well as provides assistance in creating timing constraints from the design requirements.

Input Timing Constraints

This section discusses the methodology for the specification of input timing constraints.

Overview

Input timing constraints cover the data path from the external pin of the FPGA to the internal register that captures that data. The constraint used to specify the input timing is the OFFSET IN constraint. The best method for specifying the input timing requirements depends on the type (source/system synchronous) and data rate (SDR/DDR) of the interface.

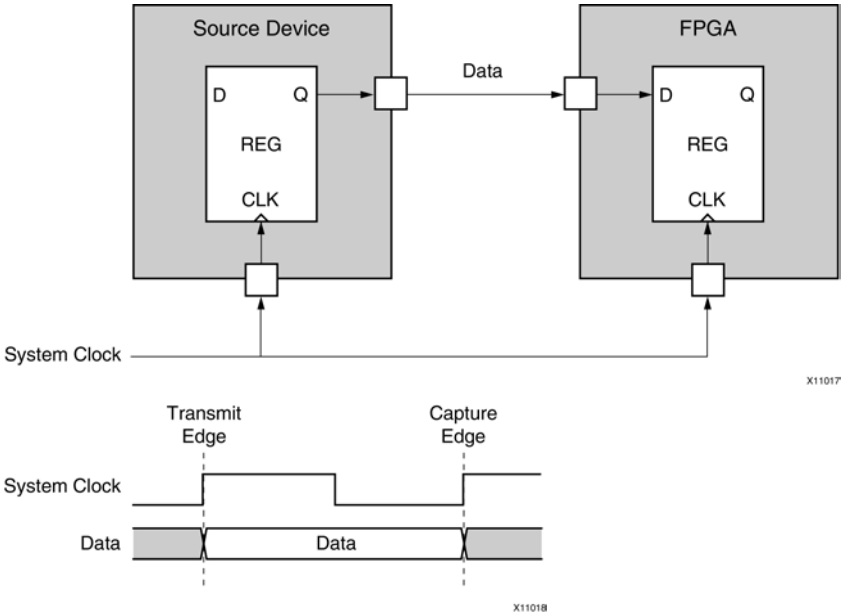
The OFFSET IN constraint defines the relationship between the data and the clock edge used to capture that data at the pins of the FPGA. When analyzing the OFFSET IN constraint, the timing analysis tools automatically take all internal factors affecting the delay of the clock and data into account. These factors include frequency and phase transformations of the clock, clock uncertainties, and data delay adjustments. In addition to the automatic adjustments, the designer may also add additional input clock uncertainty to the PERIOD constraint associated with the interface clock. For more information on the PERIOD constraint and adding INPUT_JITTER, see the PERIOD constraint section.

The OFFSET IN constraint is associated with a single input clock. By default, the OFFSET IN constraint covers all paths from the input pads of the FPGA to the internal registers that capture that data and are triggered by the specified OFFSET IN clock. This application of the OFFSET IN constraint is called the "global" method and is the most efficient way of specifying input timing.

System Synchronous Inputs

Overview:

The system synchronous interface is an interface in which a common system clock is used to both transfer and capture the data. An image of a simplified System Synchronous interface with associated SDR timing is shown below



Because this interface uses a common system clock, board trace delays and skew limit the operating frequency of the interface. The lower frequency also results in the system synchronous input interface typically being a single data rate (SDR) application. In this system synchronous SDR application example, the data is transmitted from the source device on one rising clock edge and captured in the FPGA on the next rising clock edge.

Method:

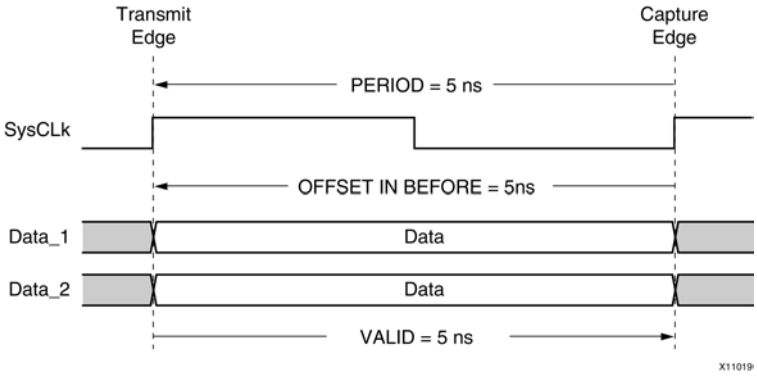
The global OFFSET IN constraint is the most efficient method of specifying the input timing for a system synchronous interface. In this method, one OFFSET IN constraint is defined for each system synchronous input interface clock. This single constraint will cover the paths of all input data bits that are captured in registers triggered by the specified input clock.

The process for specifying the input timing is:

1. Specify the clock PERIOD constraint for the input clock associated with the interface.
2. Define the global OFFSET IN constraint for the interface.

Example:

The example below shows a timing diagram for an ideal System Synchronous SDR interface. The interface has a clock period of 5 ns and the data for both bits of the bus remains valid for the entire period.



The global OFFSET IN constraint is defined as:

`OFFSET = IN` *value* **`VALID`** *value* **`BEFORE`** *clock*`;`

In the OFFSET IN constraint, the OFFSET=IN *value* determines the time from the capturing clock edge in which data first becomes valid. In this system synchronous example, the data becomes valid 5 ns prior to the capturing clock edge. In the OFFSET IN constraint, the VALID *value* determines the duration in which data remains valid. In this example, the data remains valid for 5 ns. For this example, the complete OFFSET IN specification with associated PERIOD constraint is given below:

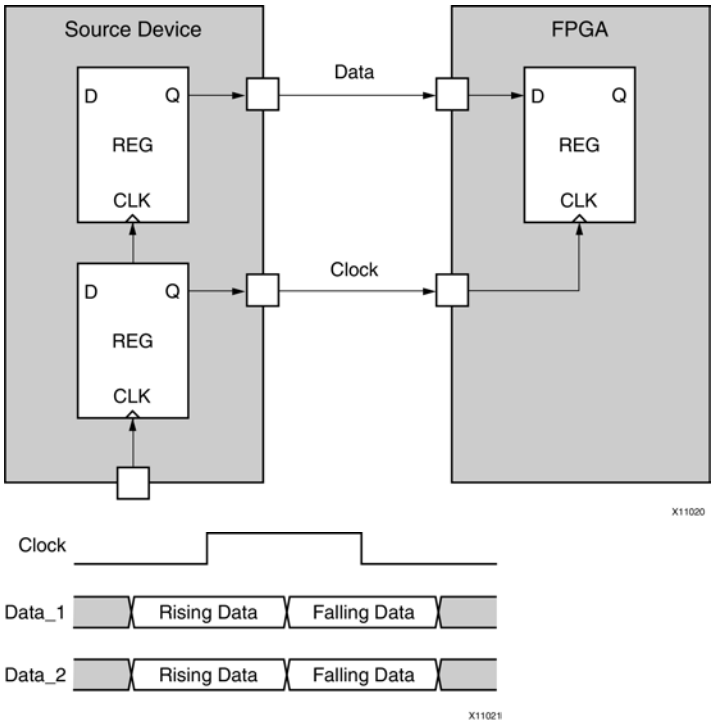
```
NET “SysClk” TNM_NET = “SysClk”;
TIMESPEC “TS_SysClk” = PERIOD “SysClk” 5 ns HIGH 50%;
OFFSET = IN 5 ns VALID 5 ns BEFORE “SysClk”;
```

This global constraint will cover both of the data bits of the bus: data1, and data2.

Source Synchronous Inputs

Overview:

The source synchronous interface is an interface in which a clock is regenerated and transmitted along with the data from the source device. This clock is then used to capture the data in the FPGA. An image of a simplified Source Synchronous interface with associated DDR timing is shown below.



Because this interface uses a regenerated clock that is transmitted along the same board traces as the data, the board trace delays and skew no longer limit the operating frequency of the interface. The higher frequency also results in the source synchronous input interface typically being a dual data rate (DDR) application. In this source synchronous DDR application example, unique data is transmitted from the source device on both the rising and falling clock edges and captured in the FPGA using the regenerated clock.

Method:

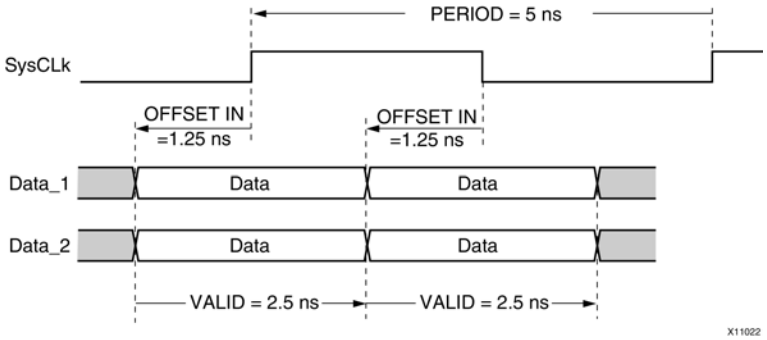
The global OFFSET IN constraint is the most efficient method of specifying the input timing for a source synchronous interface. In the DDR interface, one OFFSET IN constraints is defined for each edge of the input interface clock. These constraints will cover the paths of all input data bits that are captured in registers triggered by the specified input clock edge.

The process for specifying the input timing is:

1. Specify the clock PERIOD constraint for the input clock associated with the interface.
2. Define the global OFFSET IN constraint for the rising edge of the interface.
3. Define the global OFFSET IN constraint for the falling edge of the interface.

Example:

The example below shows a timing diagram for an ideal Source Synchronous DDR interface. The interface has a clock period of 5 ns with a 50/50 duty cycle, and the data for both bits of the bus remains valid for the entire ½ period.



The global OFFSET IN constraint for the DDR case is defined as:

OFFSET = IN *value* **VALID** *value* **BEFORE** *clock* **RISING;**

OFFSET = IN *value* **VALID** *value* **BEFORE** *clock* **FALLING;**

In the OFFSET IN constraint, the OFFSET=IN *value* determines the time from the capturing clock edge in which data first becomes valid. In this source synchronous example, the rising data becomes valid 1.25 ns prior to the capturing rising clock edge and the falling data also becomes valid 1.25 ns prior to the capturing falling clock edge. In the OFFSET IN constraint, the VALID *value* determines the duration in which data remains valid. In this example, both the rising and falling data remains valid for 2.5 ns. For this example, the complete OFFSET IN specification with associated PERIOD constraint is given below:

```
NET “SysClk” TNM_NET = “SysClk”;
TIMESPEC “TS_SysClk” = PERIOD “SysClk” 5 ns HIGH 50%;
OFFSET = IN 1.25 ns VALID 2.5 ns BEFORE “SysClk” RISING;
OFFSET = IN 1.25 ns VALID 2.5 ns BEFORE “SysClk” FALLING;
```

These global constraints will cover both of the data bits of the bus: data1, and data2.

Register-to-Register Timing Constraints

This section discusses the methodology for the specification of register-to-register synchronous path timing requirements. Register-to-register constraints cover the synchronous data paths between internal registers.

Overview

The PERIOD constraint is used to define the timing of the clock domains in the design. This constraint not only analyzes the paths within a single clock domain, but analyzes all paths between related clock domains as well. In addition, the PERIOD constraint automatically takes into account all frequency, phase, and uncertainty differences between the domains during analysis. For more information on the PERIOD constraint see the Periodconstraint section of this manual.

The application and methodology for constraining synchronous clock domains falls under several common categories. These categories include:

- Automatically related DCM/PLL/MMCM Clock Domains
- Manually related Clock Domains
- Asynchronous Clock Domains

By allowing the tools to automatically create clock relationships for DCM/PLL/MMCM output clocks, and manually defining relationships for externally related clocks, all synchronous cross-clock- domain paths will be covered by the appropriate constraints, and properly analyzed. With proper application of PERIOD constraints that follows this methodology, the need for additional cross-clock-domain constraints is eliminated.

The constraints methodology for each of these scenarios is described in the sections that follow.

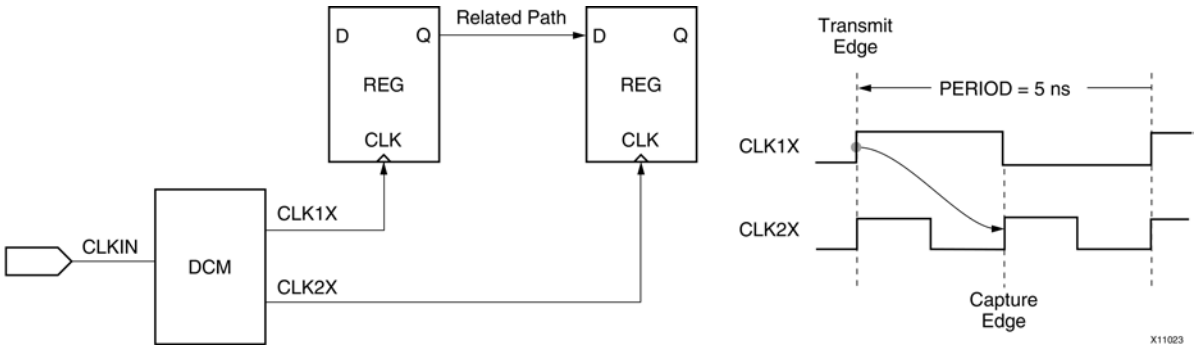
Automatically Related DCM/PLL/MMCM Clocks:

Overview:

The most common type of clock circuit is one in which the input clock is fed into a DCM/PLL/MMCM and the outputs are used to clock the synchronous paths in the device. In this scenario, the recommended methodology is to define a PERIOD constraint on the input clock to the DCM/PLL/MMCM. By placing the PERIOD constraint on the input clock, the Xilinx® tools automatically derive a new PERIOD constraint for each of the DCM/PLL/MMCM output clocks. In addition, the tools will automatically determine the clock relationships between the output clock domains, and automatically perform an analysis for any paths between these synchronous domains.

Example:

In this example the input clock of the design goes to a DCM. The circuit for this example is shown in the figure below:



The PERIOD constraint syntax for this example is defined as:

NET "*ClockName***" TNM_NET = "***TNM_NET_Name***";**

TIMESPEC "*TS_name***" = PERIOD "***TNM_NET_Name***"** *PeriodValue* **HIGH** *HighValue***%;**

In the PERIOD constraint, the PeriodValue defines the duration of the clock period. In this case, the input clock to the DCM has a period of 5 ns. The HighValue of the PERIOD constraint defines the percent of the clock waveform that is HIGH. In this example, the waveform has a 50/50 duty cycle resulting in a HighValue of 50%. The syntax for this example is given below:

```
NET "ClkIn" TNM_NET = "ClkIn";
TIMESPEC "TS_ClkIn" = PERIOD "ClkIn" 5 ns HIGH 50%;
```

Based on the input clock PERIOD constraint given above, the DCM automatically creates two output clock constraints for the DCM outputs, and automatically performs analysis between the two domains.

Manually Related Clock Domains

Overview:

In some cases the relationship between synchronous clock domains cannot be automatically determined by the tools. One example of this is when related clocks enter the FPGA device on separate pins. In this scenario, the recommended constraint methodology is to create separate PERIOD constraints for both input clocks and define a manual relationship between the clocks. Once the manual relationship is defined, all paths between the two synchronous domains are automatically analyzed with all frequency, phase, and uncertainty information automatically taken into account.

Method:

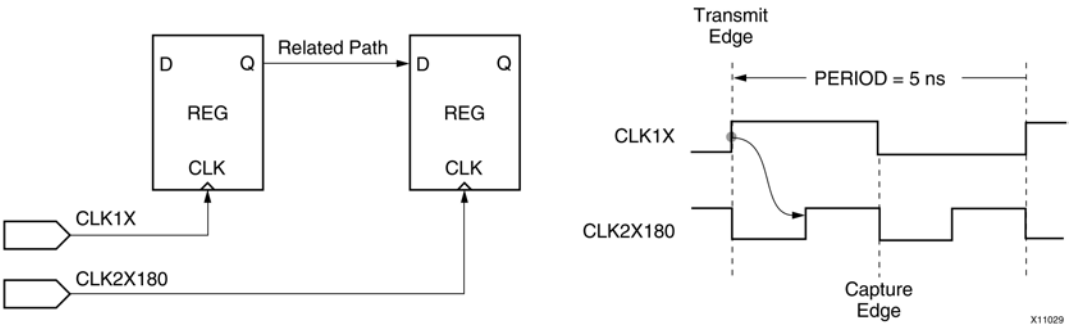
The Xilinx constraints system allows for complex manual relationships to be defined between clock domains using the PERIOD constraint. This manual relationship can include clock frequency and phase transformations. The methodology for this process is:

1. Define the PERIOD constraint for the primary clock
2. Define the PERIOD constraint for the related clock using the first PERIOD constraint as a reference

For more information on using the PERIOD constraint to define clock relationships, please see the Period constraint in this manual.

Example:

In this example two related clocks enter the FPGA device through separate external pins. The first clock, CLK1X, is the primary clock, and the second clock, CLK2X180 is the related clock. The circuit for this example is shown in the figure below:



The PERIOD constraint syntax for this example is defined as:

NET **"***PrimaryClock***" TNM_NET** = "*TNM_Primary*";

NET "*RelatedClock***" TNM_NET = "***TNM_Related***";**

TIMESPEC "*TS_primary***" = PERIOD "***TNM_Primary***"** *PeriodValue* **HIGH** *HighValue***%;**

TIMESPEC "*TS_related***" = PERIOD "***TNM_Related***"** *TS_Primary_relation* **PHASE** *value***;**

In the related PERIOD definition, the PERIOD value is defined as a time unit (period) relationship to the primary clock. The relationship is expressed in terms of the primary clock TIMESPEC. In this example CLK2X180 operates at twice the frequency of CLK1X which results in a PERIOD relationship of ½. In the related PERIOD definition, the PHASE value defines the difference in time between the rising clock edge of the source clock and the related clock. In this example, the CLK2X180 clock is 180 degrees shifted, so the rising edge begins 1.25 ns after the rising edge of the primary clock. The syntax for this example is given below:

```
NET "Clk1X" TNM_NET = "Clk1X";
NET "Clk2X180" TNM_NET = "Clk2X180";
TIMESPEC "TS_Clk1X" = PERIOD "Clk1X" 5 ns;
TIMESPEC "TS_Clk2X180" = PERIOD "Clk2X180" TS_Clk1X/2 PHASE + 1.25 ns ;
```

Asynchronous Clock Domains

Overview:

Asynchronous clock domains are defined as those in which the transmit and capture clocks bear no frequency or phase relationship. Because the clocks are not related, it is not possible to determine the final relationship for setup and hold time analysis. For this reason, it is recommended that proper asynchronous design techniques be employed to ensure the successful capture of data. However, while not required, in some cases designers wish to constrain the maximum data path delay in isolation without regard to clock path frequency or phase relationship.

Method:

The Xilinx constraints system allows for the constraining of the maximum data path delay without regard to source and destination clock frequency and phase relationship.

This requirement is specified using the FROM-TO constraint with the DATAPATHONLY keyword.

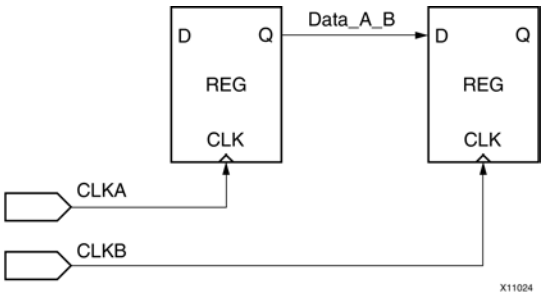
The methodology for this process is:

1. Define a time group for the source registers
2. Define a time group for the destination registers
3. Define the maximum delay of the net using the FROM-TO constraint between the two time groups with DATAPATHONLY keyword.

For more information on using the FROM-TO constraint with the DATAPATHONLY keyword, see the FROM-TO constraint section of this manual.

Example:

In this example two unrelated clocks enter the FPGA device through separate external pins. The first clock, CLKA, is the source clock, and the second clock, CLKB is the destination clock. The circuit for this example is shown in the figure below:




```
NET "CLKA" TNM_NET = FFS "GRP_A";
NET "CLKB" TNM_NET = FFS "GRP_B";
TIMESPEC TS_Example = FROM "GRP_A" TO "GRP_B" 5 ns DATAPATHONLY;
```

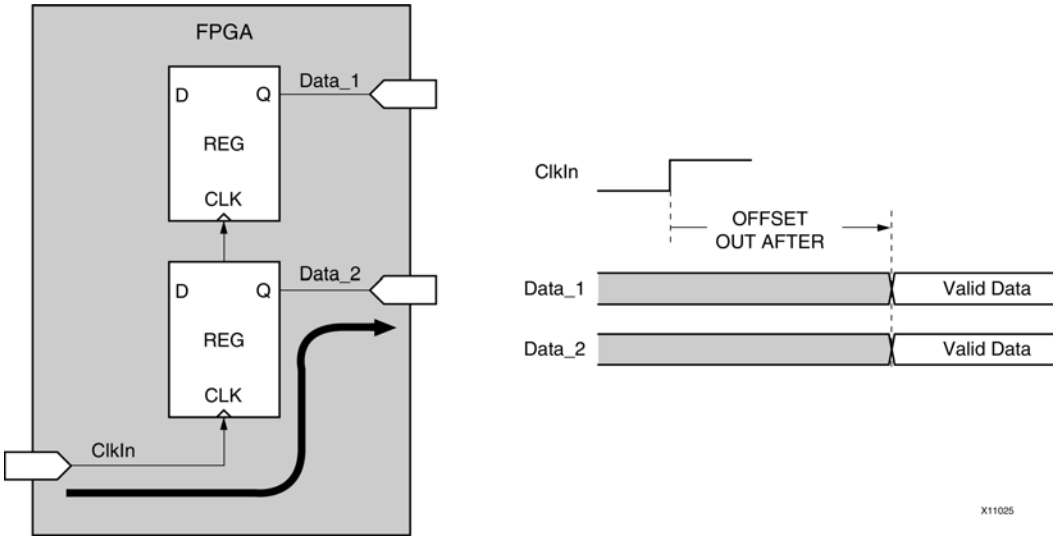
Output Timing Constraints

This section discusses the methodology for the specification of output timing constraints. Output timing constraints cover the data path from a register inside the FPGA to the external pin of the FPGA.

Overview

The constraint used to specify the output timing is the OFFSET OUT constraint. The best method for specifying the output timing requirements depends on the type (source/system synchronous) and data rate (SDR/DDR) of the interface.

The OFFSET OUT constraint defines the maximum time allowed for data to be transmitted from the FPGA. The output delay path begins at the input clock pin of the FPGA and continues through the output register to the data pins of the FPGA. This path is shown in the diagram below.

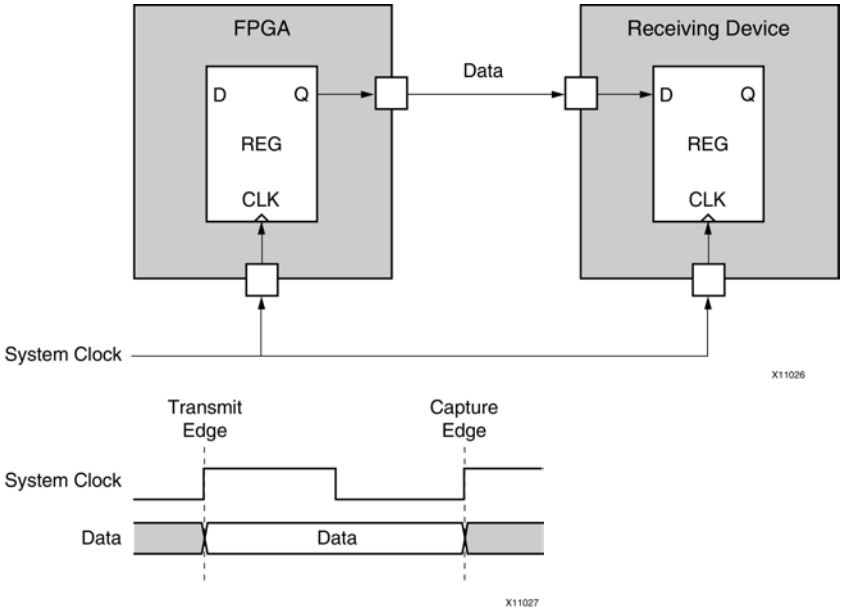


When analyzing the OFFSET OUT constraint, the timing analysis tools automatically take all internal factors affecting the delay of the clock and data into account. These factors include frequency and phase transformations of the clock, clock uncertainties, and data path delay adjustments. For more information on the OFFSET OUT constraint, please see the OFFSET OUT constraint section of this manual.

System Synchronous Output

Overview:

The system synchronous output interface is an interface in which a common system clock is used to both transfer and capture the data. An image of a simplified System Synchronous output interface with associated single data rate (SDR) timing is shown below.



Because this interface uses a common system clock, only the data will be transmitted from the FPGA to the receiving device.

Method:

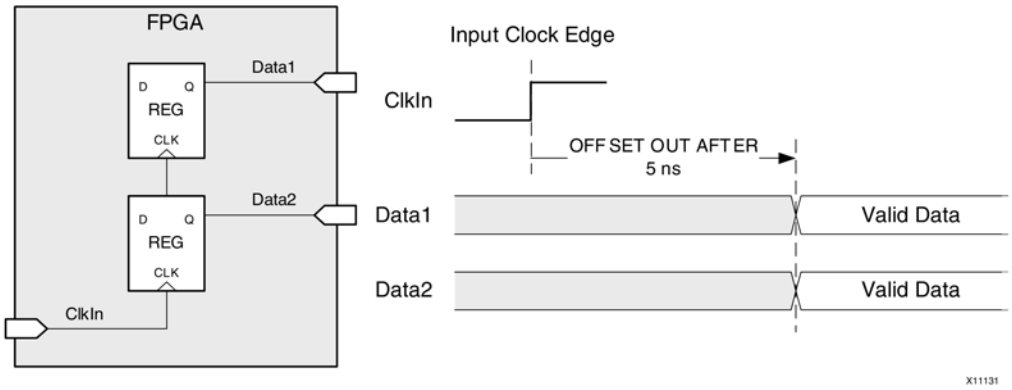
The global OFFSET OUT constraint is the most efficient method of specifying the output timing for the system synchronous interface. In the global method, one OFFSET OUT constraint is defined for each system synchronous output interface clock. This single constraint covers the paths of all output data bits sent from registers triggered by the specified output clock.

The process for specifying the output timing is:

1. Define a time name (TNM) for the output clock to create a timegroup which contains all output registers triggered by the output clock.
2. Define the global OFFSET OUT constraint for the interface

Example:

The example below shows the interface and a timing diagram for a System Synchronous SDR output interface. The data in this example must become valid at the output pins a maximum of 5 ns after the input clock edge at the pin of the FPGA.



The global OFFSET OUT constraint for the system synchronous interface is defined as:

`OFFSET = OUT`*value* **`VALID`** *value* **`AFTER`** *clock*`;`

In the OFFSET OUT constraint, the OFFSET=OUT *value*determines the maximum time from the rising clock edge at the input clock port until the data first becomes valid at the data output port of the FPGA. In this system synchronous example, the output data must become valid at least 5 ns after the input clock edge. For this example, the complete OFFSET OUT specification is given below:

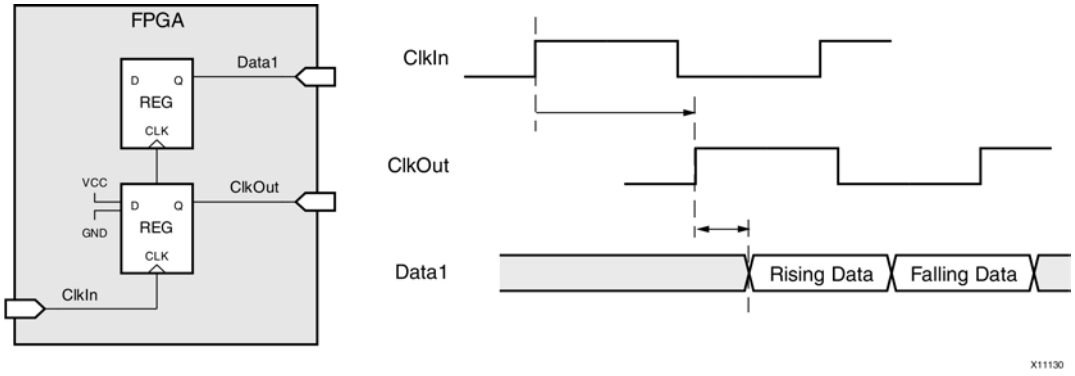
```
NET "ClkIn" TNM_NET = "ClkIn";
OFFSET = OUT 5 ns AFTER "ClkIn";
```

This global constraint covers both of the output data bits of the bus: data1 and data2.

Source Synchronous Outputs

Overview:

The source synchronous output interface is an interface in which a clock is regenerated and transmitted along with the data from the FPGA. An image of a simplified Source Synchronous output interface with associated DDR timing is shown below.



Because the regenerated clock is transmitted along with the data, the interface is primarily limited in performance by system noise and the skew between the regenerated clock and the data bits. In this interface, the time from the input clock edge to the output data becoming valid is not as important as the skew between the output data bits and in most cases can be left unconstrained.

Method:

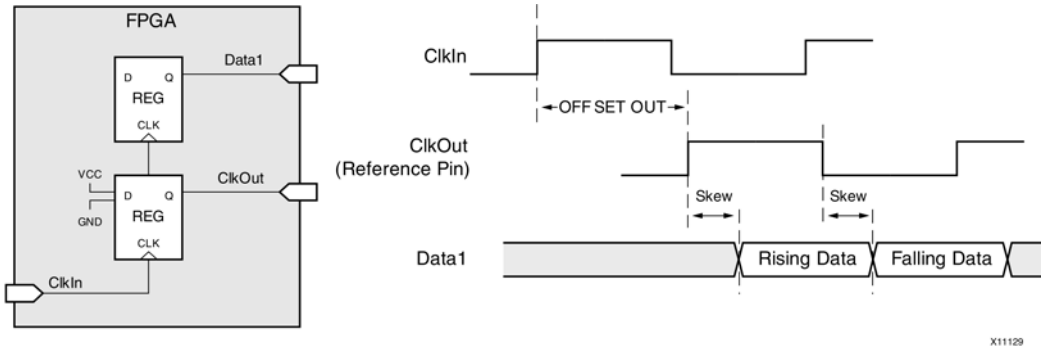
The global OFFSET OUT constraint is the most efficient method of specifying the output timing for a source synchronous interface. In the DDR interface, one OFFSET OUT constraint is defined for each edge of the output interface clock. These constraints cover the paths of all output data bits that are transmitted by registers triggered with the specified output clock edge.

The process for specifying the output timing is:

1. Define a time name (TNM) for the output clock to create a timegroup which contains all output registers triggered by the output clock
2. Define the global OFFSET OUT constraint for the rising edge of the interface
3. Define the global OFFSET OUT constraint for the falling edge of the interface

Example:

The example below shows a timing diagram for an ideal Source Synchronous DDR interface. In this interface example the absolute clock to output time is not important, and only the skew between the regenerated clock and the output data bits is desired.



The global OFFSET OUT constraints for the DDR case are defined as:

`OFFSET = OUT AFTER`*clock* **`REFERENCE_PIN "REF_CLK" RISING;`**

`OFFSET = OUT AFTER`*clock* **`REFERENCE_PIN "REF_CLK" FALLING;`**

In the OFFSET OUT constraint, the OFFSET=OUT *value*determines the maximum time from the rising clock edge at the input clock port until the data first becomes valid at the data output port of the FPGA. When the value is omitted from the OFFSET OUT constraint, the constraint becomes a report only specification which reports the skew of the output bus. The REFERENCE_PIN keyword in the constraint defines the regenerated output clock as the reference point for which the skew of the output data pins is reported against.

For this example, the complete OFFSET OUT specification for both the rising and falling clock edges is given below:

```
NET "ClkIn" TNM_NET = "ClkIn";
OFFSET = OUT AFTER "ClkIn" REFERENCE_PIN "ClkOut" RISING;
OFFSET = OUT AFTER "ClkIn" REFERENCE_PIN "ClkOut" FALLING;
```

Exception Timing Constraints

Overview

By using the global definitions of the input, register-to-register, and output timing constraints, the majority of the paths in the design will be properly constrained. However, in certain cases a small number of paths will contain exceptions to the global constraint rules. The most common type of exceptions specified are:

1. False Paths
2. Multi-Cycle Paths

False Paths

Overview:

In some cases, it may be desired to remove a set of paths from analysis if it is known that these paths do not affect the timing performance of the design.

Method:

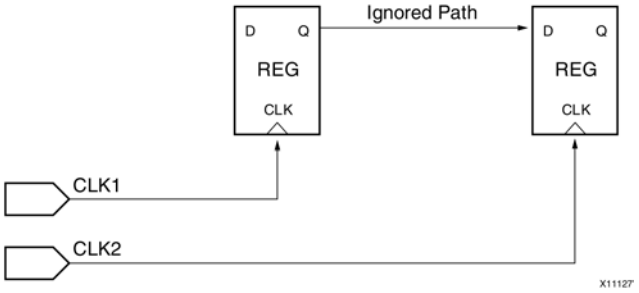
One common method of specifying the set of paths to remove from timing analysis is by using the FROM-TO constraint with the timing ignore (TIG) keyword. This method allows the designer the flexibility to specify a set of registers in a source time group, a set of registers in a destination time group, and automatically remove all paths between those time groups from analysis.

The process for specifying the timing ignore (TIG) constraint for this method is:

1. Define a set of registers for the source time group
2. Define a set of registers for the destination time group
3. Define a FROM-TO constraint with a TIG keyword to remove the paths between the groups

Example:

This example shows a hypothetical case in which a path between two registers does not affect the timing of the design, and is desired to be removed from analysis. A block diagram of the example circuit is shown in the figure below.



The generic syntax for defining a timing ignore (TIG) between time groups is:

```
TIMESPEC "TSid" = FROM "SRC_GRP" TO "DST_GRP" TIG;
```

In the FROM-TO TIG example, the SRC_GRP defines the set of source registers in which path tracing will begin from while the DST_GRP defines the set of destination registers the path tracing will end at. All paths that begin in the SRC_GRP and end in the DST_GRP will be ignored.

The specific syntax for this example is:

```
NET “CLK1” TNM_NET = FFS “GRP_1”;
NET “CLK2” TNM_NET = FFS “GRP_2”;
TIMESPEC TS_Example = FROM “GRP_1” TO “GRP_2” TIG;
```

Multi-Cycle Paths

Overview:

A multi-cycle path is a path in which data is transferred from source to destination register at a rate that is less than the clock frequency defined in the PERIOD specification. This scenario most often occurs when the registers are gated with a common clock enable signal. By defining a multi-cycle path, the timing constraints for these registers will be relaxed over the default PERIOD constraint, and the implementation tools will be able to prioritize the implementation of these paths appropriately.

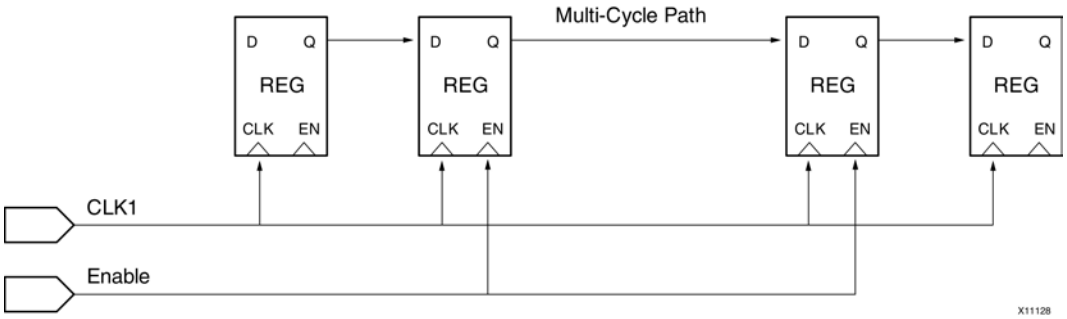
Method:

The process for specifying the FROM-TO multi-cycle constraint for this method is:

1. Define a PERIOD constraint for the common clock domain
2. Define a set of registers based on a common clock enable signal
3. Define a FROM-TO multi-cycle constraint describing the new timing requirement

Example:

This example shows a hypothetical case in which a path between two registers is clocked by a common clock enable signal. The clock enable is toggled at a rate that is one half of the reference clock. A block diagram of the example circuit is shown in the figure below.



The generic syntax for defining a multi-cycle path between time groups is:

```
TIMESPEC "TSid" = FROM "MC_GRP" TO "MC_GRP" value;
```

In the FROM-TO multi-cycle example, the MC_GRP defines the set of registers which are driven by a common clock enable signal. All paths that begin in the MC_GRP and end in the MC_GRP will have the multi-cycle timing requirement applied to them while paths into and out of the MC_GRP will be analyzed with the appropriate PERIOD specification.

The specific syntax for this example is:

```
NET "CLK1" TNM_NET = "CLK1";
TIMESPEC "TS_CLK1" = PERIOD "CLK1" 5 ns HIGH 50%;
ET "Enable" TNM_NET = FFS "MC_GRP";
TIMESPEC TS_Example = FROM "MC_GRP" TO "MC_GRP" TS_CLK1*2;
```

Chapter 6

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.
- Additional Information

  Additional information is provided for certain constraints.

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

Alphabetized List of Xilinx Constraints

This chapter contains information on the following constraints:

- Asynchronous Register (ASYNC_REG)
- BEL
- Block Name (BLKNM)
- BUFG (CPLD)
- Clock Dedicated Route
- Collapse (COLLAPSE)
- Component Group (COMPGRP)
- Configuration Mode (CONFIG_MODE)
- CoolCLOCK (COOL_CLK)
- Data Gate (DATA_GATE)
- DEFAULT (Default)
- DCI_CASCADE
- DCI_VALUE
- Directed Routing (DIRECTED_ROUTING)
- Disable (DISABLE)
- Drive (DRIVE)
- Drop Specifications (DROP_SPEC)
- Enable (ENABLE)
- Enable Suspend (ENABLE_SUSPEND)
- Fast (FAST)
- Feedback (FEEDBACK)
- File (FILE)
- Float (FLOAT)
- From Thru To (FROM-THRU-TO)
- From To (FROM-TO)
- Hierarchical Block Name (HBLKNM)
- Hierarchical Lookup Table Name (HLUTNM)
- HU_SET
- Input Buffer Delay Value (IBUF_DELAY_VALUE)
- IFD_DELAY_VALUE
- Input Registers (INREG)
- IOB
- Input Output Block Delay (IOBDELAY)
- Input Output Delay Group (IODELAY_GROUP)
- Input Output Standard (IOSTANDARD)
- Keep (KEEP)
- Keeper (KEEPER)
- Keep Hierarchy (KEEP_HIERARCHY)
- Location (LOC)
- Locate (LOCATE)
- Lock Pins (LOCK_PINS)
- Lookup Table Name (LUTNM)
- Map (MAP)

- Master Input Output Delay Group (MIODELAY_GROUP)
- Maximum Delay (MAXDELAY)
- Maximum Fanout(MAX_FANOUT)
- Maximum Product Terms (MAXPT)
- Maximum Skew (MAXSKEW)
- No Delay (NODELAY)
- No Reduce (NOREDUCE)
- Offset In (OFFSET IN)
- Offset Out (OFFSET OUT)
- Open Drain (OPEN_DRAIN)
- Optimizer Effort (OPT_EFFORT)
- Optimize (OPTIMIZE)
- 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)
- Registers (REG)
- Relative Location (RLOC)
- Relative Location Origin (RLOC_ORIGIN)
- Relative Location Range (RLOC_RANGE)
- Save Net Flag (SAVE NET FLAG)
- Schmitt Trigger (SCHMITT_TRIGGER)
- Slew (SLEW)
- Slow (SLOW)
- Stepping (STEPPING)
- Suspend (SUSPEND)
- 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)

AREA_GROUP (Area Group)

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 the Syntax section for this constraint.

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 below.

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***" GROUP=**{**OPEN**|**CLOSED**};**

or

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

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 all FPGA devices, the *RANGE* syntax is as follows:

RANGE=SLICE_X# Y#:SLICE_X#Y#

RANGE=RAMB16_X#Y#:RAMB16_X#Y#

RANGE=MULT18X18_X #Y#:MULT18X18_X#Y#

All FPGA devices SLICES are supported. If an AREA_GROUP contains both Block RAM and SLICEs, two separate AREA_GROUP RANGEs can be specified: one for BRAMs and one for SLICEs.

All locations in the FPGA are specified in terms of X and Y coordinates. You can use the wildcard character for either the X coordinate or the Y coordinate.

The RANGE value can also be specified as a CLOCK REGION or set of CLOCK REGIONs. This syntax is supported for all INST types that can be used in AREA_GROUP constraints.

For all FPGA devices, AREA_GROUP is supported for various clock regions:

For a single region:

AREA_GROUP "*groupname***" RANGE=CLOCKREGION_X#Y#;**

For a range of clock regions that form a rectangle:

AREA_GROUP "*group_name***" RANGE=CLOCKREGION_X#Y#:CLOCKREGION_X#Y#;**

For a list of clock regions:

AREA_GROUP "*groupname***" RANGE=CLOCKREGION_X#Y#,CLOCKREGION_X#Y#,…,;**

The valid X# and Y# values vary by device.

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 BRAMs, or DSP block/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.

COMPRESSION on AREA_GROUPs does not apply when Timing Driven Packing and Placement is in MAP(-timing).

COMPRESSION on AREA_GROUPs is not supported for Virtex®-5.

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.

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.

The default value is PLACE=OPEN.

AREA_GROUP Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach AREA_GROUP=*groupname* to a valid instance.
- Attach RANGE =*range* to a CONFIG symbol.
- Attach COMPRESSION=*percent* 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*, GROUP={**OPEN** |**CLOSED**}, and PLACE={**OPEN** |**CLOSED**}.
- Attribute Values: *groupname*, *range* , *percent*, GROUP={**OPEN** |**CLOSED**}, PLACE={**OPEN** |**CLOSED**}

UCF and NCF Syntax

The following example assigns all the logical blocks in state_machine_X to the area group, "group1," and places logic in the physical area between SLICE row 1, column 1 and SLICE row 10, column 10.

```
INST “state_machine_X” AREA_GROUP=group1;
AREA_GROUP “group1” RANGE=SLICE_X1Y1:SLICE_X10Y10;
```

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

Constraints Editor Syntax

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Miscellaneous in the Constraint Type list box, double-click Time Group Based Area Group to access a dialog box.

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* **;**

- *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 any DCM, PLL, or MMCM, new TNM_NET groups and PERIOD specifications are created at the DCM, PLL, or MMCM outputs. If the original TNM_NET is used to define an area group, and if more than one clock tap is used on the DCM, PLL, or MMCM, 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 DCM, PLL, or MMCM, 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

ASYNC_REG (Asynchronous Register)

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 Architecture Support

This 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 Propagation Rules

Applies to the register or latch to which it is attached

ASYNC_REG Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Miscellaneous in the Constraint Type list box, double-click Asynchronous Registers to access a dialog box.

BEL (BEL)

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 component level. Examples of components are SLICEs, BRAMs, ILOGICs, OLOGICs, and IOBs. BEL allows specification as to which particular BEL site of the component to be used. For example, this can be used to specify the specific LUT or FF to be used within a SLICE. 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 Architecture Support

The BEL constraint applies to all supported FPGA architectures.

BEL Applicable Elements

Registers	Latches
LUTs LUTRAMs RAMB18s	SRLs

BEL Propagation Rules

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

BEL Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- 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

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 the UCF and NCF Syntax for this constraint.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

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 the UCF and NCF Syntax for this constraint.

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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;
```

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

BLKNM (Block Name)

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 Architecture Support

This 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 details. For more information, see the device data sheet.

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

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 Propagation Rules

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

BLKNM Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

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

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

MODEL "*entity_name***" blknm =** *block_name***;**

BEGIN MODEL "*entity_name***"**

INST "*instance_name***" blknm =** *block_name***;**

END;

BUFG (BUFG)

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 Architecture Support

This constraint applies to CPLD devices only.

BUFG Applicable Elements

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

BUFG 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 Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- 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

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.

Verilog Syntax

Place this constraint immediately before the instantiation:

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 .

UCF and NCF Syntax

The basic UCF syntax is

NET "*net_name* **" BUFG={CLK | OE | SR | DATA_GATE};**

INST "*instance_name***" BUFG={CLK | OE | SR| DATA_GATE};**

- CLK designates a global clock pin (all CPLD families)
- OE designates a global 3-state control pin (all CPLDs except CoolRunner™-II and CoolRunner XPLA3 devices) or internal global 3-state control line (CoolRunner-II device only).
- SR designates a global set/reset pin (all CPLDs except CoolRunner-II and CoolRunner XPLA3 devices)
- 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

BEGIN MODEL "*entity_name* **"**

NET "*signal_name***" BUFG = {CLK|OE |SR|DATA_GATE} ;**

END;

CLOCK_DEDICATED_ROUTE (Clock Dedicated Route)

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 Architecture Support

The constraint applies to all supported FPGA architectures.

CLOCK_DEDICATED_ROUTE Applicable Elements

- Nets
- Input and output pins of the following primitives:
  - BUFG
  - BUFR
  - DCM
  - PLLPMCD
  - GT11
  - GT11 DUAL
  - GT11 CLK
  - GTP DUAL

CLOCK_DEDICATED_ROUTE Propagation Rules

Applies to the NET or INSTANCE PIN.

CLOCK_DEDICATED_ROUTE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: CLOCK_DEDICATED_ROUTE
- TRUE, FALSE

UCF and NCF Syntax

The syntax is:

PIN "*BEL_INSTANCE_NAME.PIN* **" CLOCK_DEDICATED_ROUTE =** {**TRUE** | **FALSE**}**;**

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

COLLAPSE Architecture Support

This constraint applies to CPLD devices only.

COLLAPSE Applicable Elements

Any internal net.

COLLAPSE Propagation Rules

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

COLLAPSE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

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

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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;
```

COMPGRP (Component Group)

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

COMPGRP Architecture Support

This constraint applies to FPGA devices only.

COMPGRP Applicable Elements

Groups of components

COMPGRP Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

PCF Syntax

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

where

comp_item is one of the following

- **COMP** "*comp_name*"
- **COMPGRP** "*group_name*"

CONFIG_MODE (Configuration Mode)

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 Architecture Support

The constraint applies to Spartan®-3, Virtex-4, and Virtex-5 devices.

CONFIG_MODE Applicable Elements

Attaches to the CONFIG symbol.

CONFIG_MODE Propagation Rules

Applies to dual-purpose I/Os

CONFIG_MODE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

The basic UCF syntax is:

CONFIG CONFIG_MODE=*string*;

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 = Slave SelectMAP Mode with Persist set to support Readback and Reconfiguration.
- S_SELECTMAP32 = Slave SelectMAP32 Mode

Note

For S_SELECTMAP32 and S_SELECTMAP32+READBACK, you can select S_SELECTMAP16 and S_SELECTMAP16+READBACK for Virtex-5 to have the right number of data pins needed persisting after configuration.

COOL_CLK (CoolCLOCK)

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 Architecture Support

This 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 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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

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

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

NET "*signal_name***" COOL_CLK;**

DATA_GATE (Data Gate)

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.

DATA_GATE Architecture Support

This constraint only applies to CoolRunner-II devices with 128 macrocells or more.

DATA_GATE Applicable Elements

I/O pads and pins

DATA_GATE Propagation Rules

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 constraint.

DATA_GATE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

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

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

NCF Syntax

Same as UCF

UCF Syntax

NET "*signal_name***" DATA_GATE;**

XCF Syntax

BEGIN MODEL "*entity_name***"**

NET "*signal_name***" data_gate={TRUE|FALSE};**

END ;

DEFAULT (Default)

This constraint allows users to set a new DEFAULT constraint value for several constraints. A specific constraint overrides the DEFAULT constraint value where applicable.

Termination is the constraint name for KEEPER, FLOAT, PULLDOWN, and PULLUP

DEFAULT Architecture Support

The DEFAULT constraint applies to the following constraints and their architectures:

- KEEPER, PULLDOWN– Applies to all FPGA devices and only the CoolRunner™-II CPLD
- PULLUP – Applies to all FPGA devices and the CoolRunner XPLA3 and CoolRunner-II CPLD devices

DEFAULT Applicable Elements

To see applicable elements for each of the constraints supported with DEFAULT, click the desired link in the basic description.

DEFAULT Propagation Rules

To see the propagation rules for each of the constraints supported with DEFAULT, click the desired link in the basic description.

DEFAULT Syntax

To see the syntax for each of the constraints supported with DEFAULT, click the desired link in the basic description.

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

The basic syntax for attaching a DEFAULT constraint to a schematic is:

- Attach to a net, instance, or pin
- Attribute Name: **DEFAULT** *constraint_name* where *constraint_name* is one of the following: KEEPER, FLOAT, PULLDOWN, PULLUP
- Attribute Values: These are determined by the *constraint_name*.

VHDL Syntax

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

attribute *attribute_name* **: string;**

Example:

```
attribute KEEPER: string;
```

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

attribute *attribute_name* **of DEFAULT is** *attribute_value***;**

Accepted *attribute_names* for DEFAULT are KEEPER, FLOAT, PULLDOWN, and PULLUP.

Accepted *attribute_values* depend on the attribute type.

Example:

```
attribute of DEFAULT KEEPER is “TRUE”;
```

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

You can specify the DEFAULT constraint as follows in Verilog code:

(* *CONSTRAINT_NAME* **= "***constrant_value***" *) DEFAULT**

The *constraint_value* is case sensitive.

Accepted *CONSTRAINT_NAMES* for the DEFAULT constraint are KEEPER, FLOAT, PULLDOWN, and PULLUP.

Accepted *constraint_values* depend on the *constraint_name*.

Example

```
(* KEEPER = “TRUE” *) DEFAULT
```

For more information on basic Verilog syntax, see Verilog.

UCF Syntax

The UCF file basic syntax for the DEFAULT constraint can be expressed as:

DEFAULT *constraint_name***;**

Accepted *constraint_names* for the DEFAULT constraint are KEEPER, FLOAT, PULLDOWN, and PULLUP.

Example:

```
DEFAULT KEEPER = TRUE;
```

XCF Syntax

The basic syntax for the DEFAULT constraint follows

BEGIN MODEL “*entity_name***”**

DEFAULT *constraint_name* [*attribute_value*] **;**

END;

Accepted *constraint_names* for the DEFAULT constraint are KEEPER, FLOAT, PULLDOWN, and PULLUP.

Accepted *attribute_values* depend on the attribute type.

Example:

```
BEGIN MODEL “my_design”
DEFAULT keeper = TRUE;
END;
```

NCF Syntax

Same as UCF

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PACE Syntax

Note PACE is only supported for CPLDs.

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.

DCI_CASCADE (DCI Cascade)

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 also uses 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 the UCF and NCF Syntax below for more rules.

DCI_CASCADE Architecture Support

This constraint applies to Virtex-5 devices only.

DCI_CASCADE Applicable Elements

A DCI_CASCADE attribute on the top level design block.

DCI_CASCADE Propagation Rules

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

DCI_CASCADE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

The UCF and NCF syntax is:

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

- *<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";
```

PCF Syntax

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

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

DCI_VALUE (DCI Value)

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 Architecture Support

The Spartan®-3 device is supported. Also Virtex®-4 and Virtex-5 devices are supported

DCI_VALUE Applicable Elements

IOBs

DCI_VALUE Propagation Rules

Applies to the IOB to which it is attached

DCI_VALUE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

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 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 with the use of LOC or RLOC constraints as well as BEL constraints.

DIRECTED_ROUTING Architecture Support

This constraint is supported for all FPGA architectures

DIRECTED_ROUTING Applicable Elements

Applies only to nets.

DIRECTED_ROUTING Propagation Rules

Not applicable

DIRECTED_ROUTING Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

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

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 by specifying RLOC constraints as well as an RLOC_ORIGIN constraint. Alternatively, location constraints (LOCs) can be specified manually by the user.

```
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 is a timing constraint that is used to turn off specific path tracing controls. A path tracing control is used to determine if a common type of path is enabled or disabled for timing analysis. 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 Architecture Support

This constraint applies to FPGA devices only.

DISABLE Applicable Elements

Global in constraints file.

DISABLE Propagation Rules

Disables timing analysis of specified block delay symbol

DISABLE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

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.

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 for Virtex®-5 and earlier architectures Enabled for Virtex-6 and Spartan®-6 architectures
reg_sr_clk	Synchronous Set/Reset to clock setup and hold checks	Enabled
lat_d_q	Data to output transparent latch delay	Disabled
lat_ce_q	Clock Enable to output transparent latch delay	Disabled
ram_we_o	RAM write enable 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

Delay Symbol Name	Path Type	Default
io_o_pad	IO output to pad propagation delay.	Enabled

PCF Syntax

Same as UCF

DRIVE (Drive)

DRIVE is a basic mapping directive that selects the output for drive strength for all supported FPGA architectures.

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

DRIVE Architecture Support

This 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 details. Also 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 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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid IOB output component
- Attribute Name: DRIVE
- Attribute Values: see the UCF and NCF Syntax for this constraint

VHDL Syntax

Place this constraint immediately before the module declaration or instantiation:

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 below for valid *values*. For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Specify the Verilog constraint as follows:

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

See the UCF and NCF Syntax below for valid *values*. For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

IOB Output Components (UCF)

For Spartan®-3 and higher devices or Virtex®-4 and higher Virtex®-4 devices:

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

12 mA is the default

SelectIO Output Components (IOBUF_SelectIO, OBUF_SelectIO, and OBUFT_SelectIO)

- For the LVTTL standard with Spartan-3 and higher devices or Virtex-4 and higher devices:

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

  12 mA is the default

- For the LVCMOS12, LVCMOS15, and LVCMOS18 standards with Spartan-3 and higher devices or Virtex-4 and higher devices:

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

  12 mA is the default

- For the LVCMOS25 and LVCMOS33 standards with Spartan-3 and higher devices or Virtex-4 and higher devices:

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

  12 mA is the default

XCF Syntax

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;

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

DROP_SPEC (Drop Specifications)

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 Architecture Support

This constraint applies to all FPGA and CPLD devices.

DROP_SPEC Applicable Elements

Timing constraints

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

TIMESPEC "TS*identifier***"=DROP_SPEC;**

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

"TS*identifier***" DROP_SPEC;**

ENABLE (Enable)

ENABLE is a timing constraint that is used to turn on specific path tracing controls. A path tracing control is used to determine if a common type of paths is enabled or disabled for timing analysis. See also Disable (DISABLE).

ENABLE Architecture Support

This constraint applies to FPGA devices only.

ENABLE Applicable Elements

Global in constraints file

ENABLE Propagation Rules

Enables timing analysis for specified path delays

ENABLE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

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

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

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 for Virtex®-5 and earlier architectures Enabled for Virtex-6 and Spartan®-6 architectures
reg_sr_clk	Synchronous Set/Reset to clock setup and hold checks	Enabled
lat_d_q	Data to output transparent latch delay	Disabled
lat_ce_q	Clock Enable to output transparent latch delay	Disabled
ram_we_o	RAM write enable 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_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

ENABLE=*delay_symbol_name* **;**

TIMEGRP *name* **ENABLE=***delay_symbol_name* **;**

ENABLE_SUSPEND (Enable Suspend)

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 Architecture Support

This 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 Propagation Rules

ENABLE_SUSPEND is a global attribute that is attached to the entire design.

ENABLE_SUSPEND Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

`CONFIG ENABLE_SUSPEND="`{**`NO`** | **`FILTERED`** | **`UNFILTERED`**}**`";`**

Example:

```
CONFIG ENABLE_SUSPEND="FILTERED";
```

FAST (Fast)

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 Architecture Support

This 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 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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: FAST
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before an instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

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

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

FEEDBACK (Feedback)

The FEEDBACK constraint is used to define the external DCM feedback path delay when the DCM is used in board deskew applications. The delay is defined as the maximum external path delay of the board trace and should not include any internal FPGA path delays. This constraint is required for the timing tools to properly determine the DCM phase shift and analyze the associated synchronous paths.

FEEDBACK Architecture Support

This constraint applies to FPGA devices only.

FEEDBACK Applicable Elements

Not applicable.

FEEDBACK Propagation Rules

Both *input_feedback_clock_net* and *output_clock_net* must correspond to pad nets. If attached to any other net, an error results. The *input_feedback_clock_net* must be an input pad and *output_clock_net* must be an output pad.

FEEDBACK Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Constraints Editor Syntax

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Miscellaneous in the Constraint Type list box, double-click DCM Feedback to access a dialog box.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

UCF Syntax

The basic UCF syntax is:

NET *output_clock_net* **FEEDBACK =** *value units* **NET** *input_feedback_clock_net***;**

- *input_feedback_clock_net* 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_clock_net* is the name of the output pad net driven by the DCM

XCF Syntax

BEGIN MODEL "*entity_name***"**

NET *output_clock_net* **FEEDBACK =** *value units* **NET** *input_feedback_clock_net***;**

END;

- *input_feedback_clock_net* 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_clock_net* is the name of the output pad net driven by the DCM

PCF Syntax

{**BEL** |**COMP**} *output_clock_net* **FEEDBACK =** *value units* {**BEL** |**COMP**} *input_feedback_clock_net***;**

FILE (File)

When you instantiate a module that resides in another netlist, NGCBuild 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 NGCBuild 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 Architecture Support

This constraint applies to all FPGA and CPLD devices.

FILE Applicable Elements

Instance declaration where the definition is defined in the specified file.

FILE Propagation Rules

Applicable only on instances

FILE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: FILE
- Attribute Values: *file_name*.*extension*

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

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* XIL_FILE = "*file_name***" *)**

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

INST <instance definition> **FILE**= <filename definition is located in>**;**

No valid syntax for UCF.

FLOAT (Float)

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 Architecture Support

This constraint can be applied to the CoolRunner™ XPLA3 and CoolRunner-II devices.

FLOAT Applicable Elements

Applies to nets or pins.

FLOAT Propagation Rules

Applies to the net or pin to which it is attached.

FLOAT Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: FLOAT
- Attribute Value: None required. TRUE, FALSE. If attached, TRUE is assumed.

VHDL Syntax

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

Place this constraint immediately before an instantiation:

Specify the Verilog constraint as follows:

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

UCF and NCF Syntax

The basic UCF syntax is:

NET "*signal_name* **" FLOAT;**

XCF Syntax

BEGIN MODEL "*entity_name***"**

NET "*signal_name***" FLOAT;**

END;

FROM-THRU-TO (From Thru To)

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 Architecture Support

This constraint applies to FPGA devices only.

FROM-THRU-TO Applicable Elements

Predefined and user-defined groups

FROM-THRU-TO Propagation Rules

Applies to the specified FROM-THRU-TO path only.

FROM-THRU-TO Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

TIMESPEC "TS*identifier"***=FROM "***source_group***" THRU "***thru_pt1***"**...[**THRU "***thru_pt2***"** ...] **TO "***destination_group***"** *value* [*Units*] [**DATAPATHONLY**];

- *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, 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

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

1. Select **Timing Constraints** > **Exceptions** > **Paths in the Constraint Type**, to bring up the Path Exceptions dialog.
2. Expand the Through points section of the dialog by clicking on the + button, in order to identify the available throught points.
3. Fill out the Path Exceptions dialog.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PCF Syntax

TS*name***=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 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 the Virtex®-5 device, the FROM-TO constraint controls both setup and hold paths.

FROM-TO Architecture Support

This constraint applies to all FPGA and CPLD devices.

FROM-TO Applicable Elements

Predefined and user-defined groups

FROM-TO Propagation Rules

Applies to a path specified between two groups.

FROM-TO Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

TIMESPEC TS*name***=FROM "***group1***" TO "***group2***"** *value* **[*DATAPATHONLY*];**

- 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

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

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Exceptions, which is under Timing Constraints, in the Constraint Type list box, double-click Paths to access a dialog box.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PCF Syntax

TS*name*`=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.

HBLKNM (Hierarchical Block Name)

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, NGCBuild 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 Architecture Support

This 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 details. Also for more information, see the devicedata sheet.

1. Registers
2. I/O elements and pads
3. FMAP
4. PULLUP
5. ACLK, GCLK
6. BUFG
7. BUFGS, BUFGP
8. ROM
9. RAMS and RAMD
10. Carry logic primitives

You can also attach HBLKNM to the net connected to the pad component in a UCF file. NGCBuild 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 Propagation Rules

When attached to a design element, HBLKNM is propagated to all applicable elements in the hierarchy within the design element. However, when attached to a NET, HBLKNM is only propagated to PADs.

HBLKNM Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: HBLKNM
- Attribute Values: *block_name*

VHDL Syntax

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***";**

block_name is a valid block name for that type of symbol

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* HBLKNM = "*block_name***" *)**

block_name is a valid block name for that type of symbol

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

The basic UCF syntax is:

NET "*net_name***" HBLKNM=***property_value***;**

INST "*instance_name***" HBLKNM=***block_name***;**

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.

HLUTNM (Hierarchical Lookup Table Name)

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 HBLKNM (Hierarchical Block Name) constraint.

HLUTNM Architecture Support

This 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, SRL16) 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) in conjunction with a 5-input read-only symbol (LUT5) 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 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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid element or symbol type
- Attribute Name: HLUTNM
- Attribute Values: <user_defined>

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* HLUTNM = "*string_value***" *)**

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 .

UCF and NCF Syntax

INST "*symbol_name***" HLUTNM=***string_value* **;**

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

MODEL "*symbol_name***" hlutnm =** *string_value* **;**

HU_SET (HU Set)

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.

NGCBuild 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 the Relative Location (RLOC) constraint.

HU_SET Architecture Support

This 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 details. Also for more information, see the device data sheet.

1. Registers
2. FMAP
3. Macro Instance
4. ROM
5. RAMS, RAMD
6. MULT18X18S
7. RAMB4_Sm_Sn, RAMB4_Sn
8. RAMB16_*Sm_Sn, RAMB16_Sn*
9. RAMB16
10. DSP48

HU_SET Propagation Rules

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

HU_SET Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: HU_SET
- Attribute Values: *set_name*

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before an instantiation:

Specify the Verilog constraint as follows:

(* HU_SET = "*set_name*" ***)**

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

The basic UCF syntax is:

INST "*instance_name***" HU_SET=***set_name* **;**

set_name is the identifier for the set

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

MODEL "*entity_name***" hu_set={yes | no};**

BEGIN MODEL "*entity_name***"**

INST "*instance_name***" hu_set=yes;**

END;

IBUF_DELAY_VALUE (Input Buffer Delay Value)

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 Architecture Support

This constraint applies to Spartan®-3A and Spartan-3E devices.

IBUF_DELAY_VALUE Applicable Elements

Any top-level I/O port.

IBUF_DELAY_VALUE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach a new property to the top-level port of the schematic
- Attribute Name-IBUF_DELAY_VALUE
- Attribute Values: 0-16

VHDL Syntax

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***";**

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

Attach a Verilog attribute to the appropriate top-level port

(* IBUF_DELAY_VALUE="*value***" *) input** *top_level_port_name*;

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

The basic UCF syntax is:

NET "*top_level_port_name***" IBUF_DELAY_VALUE =** *value*;

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)

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 Architecture Support

Spartan®-3A and Spartan-3E are supported.

IFD_DELAY_VALUE Applicable Elements

Any top-level I/O port

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net
- Attribute Name-IFD_DELAY_VALUE
- Attribute Values-0-8, AUTO

VHDL Syntax

Attach a VHDL attribute to the appropriate top-level port

```
attribute IFD_DELAY_VALUE : string;
```

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

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

The basic UCF syntax is:

NET "*top_level_port_name***" IFD_DELAY_VALUE =** *value*;

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;
```

INREG (Input Registers)

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 Architecture Support

CoolRunner XPLA3 and CoolRunner-II devices are supported.

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 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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a register, latch, or net
- Attribute Name: INREG
- Attribute Values: None (TRUE by default)

UCF Syntax

NET "*signal_name*" **INREG;**

INST "*register_name*" **INREG;**

IOB (IOB)

IOB is a basic mapping and synthesis constraint. It indicates which flip-flops and latches can be moved into the IOB/ILOGIC/OLOGIC. The mapper supports a command line option (-pr i | o | b | off) 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 Architecture Support

This constraint applies to FPGA devices only.

IOB Applicable Elements

- Non-INFF/OUTFF flip-flop and latch primitives
- Registers

IOB Propagation Rules

Applies to the design element to which it is attached

IOB Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a flip-flop or latch instance or to a register
- Attribute Name: IOB
- Attribute Values: TRUE, FALSE, AUTO, FORCE

VHDL Syntax

Declare the VHDL constraint as follows:

attribute iob: *string*;

Specify the VHDL constraint as follows:

attribute iob of {*component_name* |*entity_name*|*label_name*|*signal_name*} **:** {**component**|**entity**|**label**|**signal**} **is "**{**TRUE**|**FALSE**|**AUTO**|**FORCE**}**";**

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
- **FORCE** requires that the flip-flop or latch be pulled into an IOB, otherwise an error is given. FORCE will only produce an error if the register has I/O connections and cannot be packed in the IOB.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* IOB = "{**TRUE**|**FALSE** |**AUTO**|**FORCE**}**" *)**

See value definitions in VHDL Syntax above.

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

The basic syntax is:

INST "*instance_name***" IOB=**{**TRUE**|**FALSE**|**FORCE**};**
See value definitions in VHDL Syntax above.

The following statement instructs the mapper to place the foo/bar instance into an IOB component.

INST "foo/bar" IOB=TRUE;

XCF Syntax

BEGIN MODEL "*entity_name***"**

NET "*signal_name***" iob=**{**true**|**false**|**auto**|**force**};**

INST " *instance_name***" iob=**{**true** |**false** |**auto**|**force**};**

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

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Miscellaneous in the Constraint Type list box, double-click Registers to be Placed on IOBs to access a dialog box.

IOBDELAY (Input Output Block Delay)

IOBDELAY is a basic mapping constraint. It specifies how the input path delay elements in all 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. Xilinx® 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).

IOBDELAY Architecture Support

This constraint applies to FPGA devices only.

IOBDELAY Applicable Elements

Any I/O symbol (I/O pads, I/O buffers, or input pad nets)

IOBDELAY Propagation Rules

Although IOBDELAY is attached to an I/O symbol, it applies to the entire I/O component.

IOBDELAY Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- 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

Place this constraint immediately before the module declaration or instantiation:

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.

Verilog Syntax

Specify the Verilog constraint as follows:

`(* IOBDELAY =` {**`NONE`**|**`BOTH`**|**`IBUF`**|**`IFD`**} **`*)`**

For more information on basic Verilog syntax, see Verilog .

UCF and NCF Syntax

The basic UCF syntax is:

`INST “` *instance_name*`”` **`IOBDELAY=`**{**`NONE`**|**`BOTH`**|**`IBUF`**|**`IFD`**}`;`

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
```

IODELAY_GROUP (IODELAY Group)

IODELAY_GROUP is a design implementation constraint that groups a set of IDELAY/IODELAYs with an IDELAYCTRL to enable automatic replication and placement of IDELAYCTRL(s) in a design. For more information on the use case scenarios for this constraint, please see the IDELAYCTRL section of the appropriate architecture user’s guide.

IODELAY_GROUP Architecture Support

The IODELAY_GROUP constraint is applicable for the Virtex®-4 and Virtex-5 architectures. For Virtex-4, this constraint is only supported when using the Timing Driven Pack and Placement Option in MAP.

IODELAY_GROUP Applicable Elements

IDELAY, IODELAY, and IDELAYCTRL primitive instantiations.

IODELAY_GROUP Propagation Rules

IODELAY_GROUP is a constraint that can only be attached to a design element. It is illegal to attach IODELAY_GROUP to a net, signal, or pin. To merge two or more embedded IODELAY_GROUP constraints in your design, please see MIODELAY_GROUP.

IODELAY_GROUP Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax

Declare the VHDL constraint as follows:

```
attribute IODELAY_GROUP: string;
```

Specify the VHDL constraint as follows:

attribute IODELAY_GROUP of {*component_name*|*label_name*}**:** {**component**|**label**} **is "***group_name***";**

For a description of *group_name*, see the UCF Syntax for this constraint.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* IODELAY_GROUP = "*group_name***" *)**

For a description of *group_name*, see the UCF Syntax for this constraint.

For more information on basic Verilog syntax, see Verilog.

UCF Syntax

The basic UCF syntax for defining an IODELAY_GROUP follows:

INST "*instance_name***" IODELAY_GROUP =** *group_name***;**

where *group_name* is the name assigned to a set of IDELAY/IODELAYs and an IDELAYCTRL to uniquely define the group.

IOSTANDARD (Input Output Standard)

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®-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-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.

The CPLD fitter automatically groups outputs with compatible IOSTANDARD settings into the same bank when no location constraints are specified.

IOSTANDARD Architecture Support

The IOSTANDARD constraint applies to all FPGAs, For CPLDs, the constraint is supported for CoolRunner-II devices.

IOSTANDARD Applicable Elements

To see which design elements can be used with which device families, see the Xilinx Libraries Guides for details. Also for more information, see the device data sheet.

- IBUF, IBUFG, OBUF, OBUFT
- IBUFDS, IBUFGDS, OBUFDS, OBUFTDS
- Output Voltage Banks

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to an I/O primitive
- Attribute Name: IOSTANDARD
- Attribute Values: *iostandard_name*

For more information, see the UCF and NCF Syntax for this constraint.

VHDL Syntax

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 the UCF and NCF Syntax for this constraint.

For CPLD devices you can also apply IOSTANDARD to the pad signal.

For more information on basic VHDL syntax, see VHDL .

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

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.

UCF and NCF Syntax

The basic syntax is:

INST " *instance_name***" IOSTANDARD=** *iostandard_name***;**

NET "*pad_net_name***" IOSTANDARD=***iostandard_name***;**

iostandard_name is an IO Standard name as specified in the device data sheet

XCF Syntax

```
BEGIN MODEL "entity_name "
INST "instance_name" iostandard=string ;
NET "signal_name" iostandard=string ;
END;
```

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PACE Syntax

PACE is mainly used to assign location constraints to IOs for CPLDs only. 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. PACE is supported only for CPLDs not FPGAs.

KEEP (Keep)

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. In addition to TRUE and FALSE, synthesis (XST) accepts and additional SOFT value that instructs the tool to preserve the designated net, but also prevents it from attaching a NOMERGE constraint to this net in the synthesized netlist. As a result, the net is preserved during synthesis, but implementation tools are given all freedom to handle it. Conceptually, you are specifying a KEEP=TRUE for synthesis only, but a KEEP=FALSE for implementation tools.

KEEP Architecture Support

This constraint applies to all FPGA and CPLD devices.

KEEP Applicable Elements

Signals

KEEP Propagation Rules

Applies to the signal to which it is attached

KEEP Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net
- Attribute Name: KEEP
- Attribute Values: TRUE, FALSE, SOFT (XST only)

VHDL Syntax

Declare the VHDL constraint as follows:

```
attribute keep :  string;
```

Specify the VHDL constraint as follows:

attribute keep of *signal_name*: **signal is "**{**TRUE**|**FALSE**|**SOFT**}**";**

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before an instantiation:

Specify the Verilog constraint as follows:

(* KEEP = "{**TRUE**|**FALSE** |SOFT}**" *)**

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

The following statement ensures that the net $SIG_0 remains visible.

NET "$1I3245/$SIG_0" KEEP;

XCF Syntax

BEGIN MODEL "*entity_name***"**

NET "*signal_name***" keep=**{**yes**|**no**|**true**|**false**}**;**

END;

Note

In an XCF file, the value of the KEEP constraint may optionally be enclosed in double quotes. For the SOFT value, this becomes mandatory, as shown below:

BEGIN MODEL "*entity_name***"**

NET "*signal_name***"keep="soft";**

END;

KEEP_HIERARCHY (Keep Hierarchy)

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**

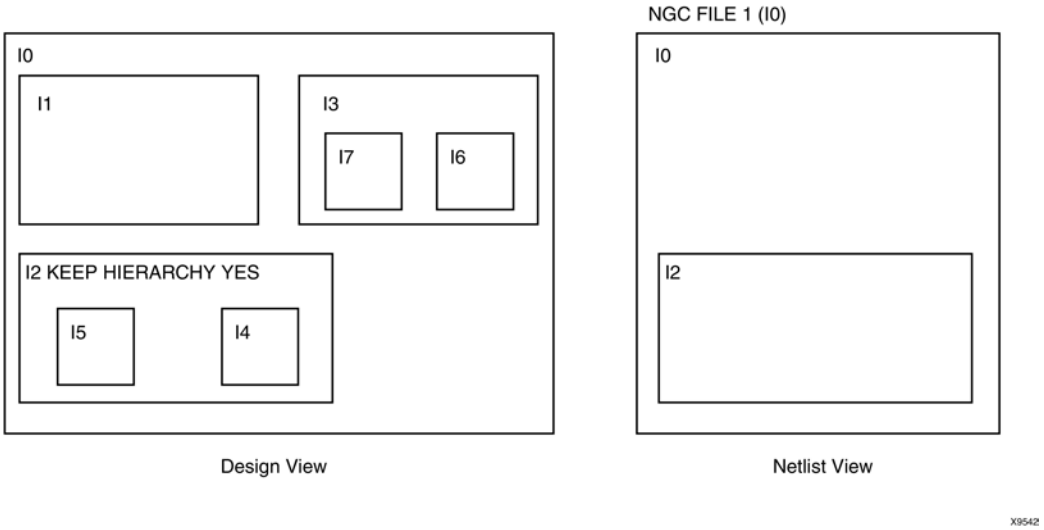
Note In XST, the KEEP_HIERARCHY constraint can be set to the following values: **`yes`**, **`true`**, **`no`**, **`false`**, and **`soft`**. When used at the command line, only **`yes`**, **`no`**, and **`soft`** are accepted.

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.

KEEP_HIERARCHY EXAMPLE



KEEP_HIERARCHY Architecture Support

>

This 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 Propagation Rules

Applies to the entity or module to which it is attached

KEEP_HIERARCHY Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to the entity or module symbol
- Attribute Name: KEEP_HIERARCHY
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* KEEP_HIERARCHY = "{TRUE|FALSE|SOFT}" *)

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

For instances:

INST "*instance_name***" KEEP_HIERARCHY={TRUE|FALSE|SOFT};**

XCF Syntax

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

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. Select **Advanced** from the **Property display level** field.

KEEPER (Keeper)

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* for more information.

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 Architecture Support

The KEEPER constraint applies to all FPGA devices and only the CoolRunner-II CPLD.

KEEPER Applicable Elements

Tristate input/output pad nets

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to an output pad net
- Attribute Name: KEEPER
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before an instantiation:

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

For more information on basic Verilog syntax, see Verilog .

UCF and NCF Syntax

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

BEGIN MODEL "*entity_name***"**

NET "*signal_name***" keeper=**{**yes|no|true|false**}**;**

```
END;
```

LOC (Location)

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 PlanAhead or the FPGA Editor 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, 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.

FPGA 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.

Following are examples of how to specify the slices in the XY coordinate system.

Single LOC Constraint Examples

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

FPGA block RAMs, and multipliers have their own specification different from the SLICE specifications. Therefore, the location value must start with "SLICE," "RAMB," or "MULT." 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.

The location values for global buffers (BUFGs) and DCM elements is the specific physical site names for available locations.

Pin assignment using the LOC constraint is not supported for bus pad symbols such as OPAD8.

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®-3, Spartan-3A, and Spartan-3E devices
	LB, RB, LT, RT, BR, TR, BL, TL	Applies to IOBs and indicates half edges (for example, left bottom, right bottom) for Spartan-3, Spartan-3A, and Spartan-3E devices

Element Types	Location Examples	Meaning
	Bank0, Bank1, Bank2, Bank3, Bank4, Bank5, Bank6, Bank7	Applies to IOBs and indicates half edges (banks) for all FPGAs
Slices	SLICE_X22Y3	SLICE_X22Y3 Slice location for all FPGAs
Block RAMs	RAMB16_X2Y56	Block RAM location for Spartan-3, Spartan-3A, Spartan-3E, and Virtex®-4 devices
	RAMB36_X2Y56	Block RAM location for Virtex-5 devices
Multipliers	MULT18X18_X#Y#	Multiplier location for Spartan-3 and Spartan-3A, devices
	DSP48a_X#Y#	Multiplier location for Spartan-3A DSP
	DSP48_X#Y#	Multiplier location for Virtex-4 and Virtex-5 devices
Digital Clock Manager	DCM_X#Y#	Digital Clock Manager for Spartan-3, Spartan-3A, and Spartan-3E devices
	DCM_ADV_X#Y#	Digital Clock Manager for Virtex-4 and Virtex-5 device
Phase Lock Loop	PLL_ADV_X#Y#	Phase Lock Loop for all FPGAs

The wildcard character (*) can be used to replace a single location with a range as shown in the following example:

SLICE_X*Y5	Any slice of a FPGA device whose Y coordinate is 5

The following is *not* supported.

Wildcard character for an FPGA global buffer, global pad, or DCM locations.

LOC Description for CPLD Devices

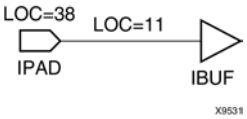
For CPLD devices, use the LOC=*pin_name* constraint on a PAD symbol or pad net to assign the signal to a specific pin. The PAD symbols are IPAD, OPAD, IOPAD, and UPAD. You can use the LOC=FBnn constraint on any instance or its output net to assign the logic or register to a specific function block or 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.

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.

LOC Priority Example



LOC Architecture Support

This 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 details. Also for more information, see the device data sheet.

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

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.

Constraint (UCF Syntax)	Devices	Description
`INST "`*instance_name*`" LOC=P12;`		Place I/O at location P12.
`INST "`*instance_name*` " LOC=SLICE_X3Y2;`	Spartan-3, Spartan-3A, Spartan-3E, Virtex®-4, and Virtex-5 devices	Place logic in slice X3Y2 on the XY SLICE grid.
`INST "`*instance_name*` " LOC=RAMB16_X0Y6;`	Spartan-3, Spartan-3A, Spartan-3E, and Virtex-4 devices	Place the logic in the block RAM located at RAMB16_X0Y6 on the XY RAMB grid.
`INST "`*instance_name*` " LOC=MULT18X18_X0Y6;`	Spartan-3 and Spartan-3A devices	Place the logic in the multiplier located at MULT18X18_X0Y6 on the XY MULT grid.
`INST "`*instance_name*` " LOC=FIFO16_X0Y15;`	Virtex-4	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 devices	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.

Multiple LOC Constraint Examples

Constraint	Devices	Description
`INST` *"instance_name* `" LOC=SLICE_X2Y10, SLICE_X1Y10;`	FPGAs	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-4, and Virtex-5 devices, specify the upper left and lower right corners of an area in which logic is to be placed. For FPGA 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.

LOC Range Constraint Examples

Constraint	Devices	Description
`INST` *"instance_name "* `LOC=SLICE_X3Y5:SLICE_X5Y20;`	FPGAs	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

The two constraint formats for **FB***ff* and **FB***ff_mm* are only applicable for outputs and bidirectional pins, not for inputs.

The first constraint format:

INST "*instance_name***" LOC=***pin_name***;**

is applicable for all types of IO.

LOC Syntax

For examples of legal placement constraints for each type of logic element in FPGA designs, see Syntax for FPGA Devices for this constraint, 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.

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to an instance
- Attribute Name: LOC
- Attribute Values: *value*

For valid values, see Syntax for FPGA Devices and Syntax for CPLD Devices for this constraint.

VHDL Syntax

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 Syntax for FPGA Devices and Syntax for CPLD Devices for this constraint.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before an instantiation:

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 Syntax for FPGA Devices and Syntax for CPLD Devices for this constraint.

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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=SLICE_X*Y8;
```

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

BEGIN MODEL " *entity_name*"

PIN "*signal_name*" **loc=***string* **;**

INST "*instance_name*" **loc=***string* **;**

END;

PCF Syntax

LOC writes out a LOCATE constraint to the PCF file. For more information, see the Locate (LOCATE) constraint.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PACE Syntax

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. PACE is supported only for CPLDs not FPGAs.

Digital Clock Manager (DCM) Constraint Examples

This section applies to all FPGAs.

You can lock the DCM in the UCF file. The syntax is as follows:

INST "*instance_name***" LOC = DCM_X** *A***Y***B***;** (for all Spartan devices)

INST "*instance_name***" LOC = DCM_ADV_X** *A***Y***B***;** (for Virtex-4, and Virtex-5 devices)

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.

Slice-Based XY Grid Designations

Spartan-3 and higher and Virtex-4 and higher architectures are the only ones 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*;

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 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 SLICE at row 7, column 3.

Schematic	LOC=SLICE_X7Y3
UCF	INST "$1I323"LOC=SLICE_X2Y4, SLICE_X3Y4;

Example Two

Place the FMAP symbol in either the SLICE at row 2, column 4 or the SLICEat row 3, column 4.

Schematic	LOC=SLICE_X2Y4, SLICE_X3Y4
UCF	INST “top/dec0011” LOC=CLB_R2C4,CLB_R3C4;

Example Three

Place the FMAP symbol in the area bounded by SLICE X5Y5 in the upper left corner and SLICE X10Y8 in the lower right

Schematic	LOC=SLICE_X5Y5:SLICE_X10Y8
UCF	INST “$3I27” LOC=SLICE_X5Y5:SLICE_X10Y8;

Multiplier Constraint Examples

This section applies to FPGAs.

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.

An FPGA multiplier has a different XY grid specification than slices and block RAMs. Spartan-3, Spartan-3A, Spartan-3E, are specified using MULT18X18_X#Y#, Spartan-3A DSP is specified using DSP48a_X#Y#, and Virtex-4, Virtex-5 are specified using DSP48_X#Y#, 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.

Slice-Based XY Designations

Spartan-3 and higher and Virtex-4 and higher 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 FPGAs

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-3 and Higher Devices

An FPGA block RAM has a different XY grid specification than a slice or multiplier. 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 (for all FPGA except Virtex-5) LOC=RAMB36_X0Y0 (for Virtex-5)
UCF	INST "/top-7/rq" LOC=RAMB16_X0Y0;

Slice Constraint Examples

This section applies to all FPGAs 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.

INST "*instance_name***" LOC=SLICE_X133Y10;**

Slice Constraints Example Two

The following UCF statement places logic within the first column of slices. 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. 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

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*;

Slices Prohibited 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 is a basic placement constraint that specifies a single location, multiple single locations, or a location range.

LOCATE Architecture Support

This constraint applies to FPGA devices only.

LOCATE Applicable Elements

- CLBs
- IOBs
- DCMs
- Clock logic
- Macros

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

PCF Syntax

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)

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 Architecture Support

This 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 Propagation Rules

LOCK_PINS is applied only to a single LUT instance.

LOCK_PINS Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax

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 .

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

```
(* LOCK_PINS = “all” *)
```

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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;
```

LUTNM (Lookup Table Name)

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 BLKNM (Block Name) constraint.

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 Propagation Rules

The LUTNM constraint can be applied to two symbols that are unique within the design.

LUTNM Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid element or symbol type
- Attribute Name: LUTNM
- Attribute Value: <user_defined>

VHDL Syntax

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***";**

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.

Verilog Syntax

Place the following attribute specification before the port declaration in the top-level Verilog code:

(* LUTNM = "*value***" *)**

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.

UCF/NCF Syntax

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 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 Architecture Support

This constraint applies to FPGA devices only.

MAP Applicable Elements

FMAP

MAP Propagation Rules

Applies to the design element to which it is attached

MAP Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

The basic UCF syntax is:

INST " *instance_name***" MAP=[PUC | PUO | PLC | PLO] ;**

the terms have the following meanings:

PUC

The CLB pins are unlocked (U) and the CLB is closed (C). The software *can* swap signals among the pins on the CLB, but *cannot* add or remove logic from the CLB.

PUO

The CLB pins are unlocked (U) and the CLB is open (O). The software *can* swap signals among the pins on the CLB, and *can* add or remove logic from the CLB.

PLC

The CLB pins are locked (L) and the CLB is closed (C). The software *cannot* swap signals among the pins on the CLB, and *cannot* add or remove logic from the CLB.

PLO

The CLB pins are locked (L) and the CLB is open (O). The software *cannot* swap signals among the pins on the CLB, but *can* add or remove logic from the CLB.

The default is PUO. Currently, only PUC and PUO are observed. PLC and PLO are translated into PUC and PUO, respectively.

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;
```

MAX_FANOUT (Max Fanout)

Max Fanout (MAX_FANOUT) limits the fanout of nets or signals. Depending on the value of the constraint, both XST and MAP will limit the fanout of a net when this constraint is applied. The value can either be an integer (XST only) or REDUCE (MAP only).

MAX_FANOUT for XST

Default integer values for XST are shown in the following table. Max Fanout is both a global and a local constraint in XST.

Max Fanout Default Values Devices

Devices	Default Value
Spartan®-3, Spartan-3E, Spartan-3A, Spartan-3A D	500
Virtex®-4	500
Virtex-5	100000 (One Hundred Thousand)

Large fanouts can cause routability problems. XST tries to limit fanout by duplicating gates or by inserting buffers. This limit is not a technology limit but a guide to XST. It may happen that this limit is not exactly respected, especially when this limit is small (less than 30).

In most cases, fanout control is performed by duplicating the gate driving the net with a large fanout. If the duplication cannot be performed, buffers are inserted. These buffers are protected against logic trimming at the implementation level by defining a Keep (KEEP) attribute in the NGC file. If the register replication option is set to **no**, only buffers are used to control fanout of flip-flops and latches.

Max Fanout is global for the design, but you can control maximum fanout independently for each entity or module or for given individual signals by using constraints.

If the actual net fanout is less than the Max Fanout value, XST behavior depends on how Max Fanout is specified.

- If the value of Max Fanout is set in ISE® Design Suite in the command line, or is attached to a specific hierarchical block, XST interprets its value as a guidance.
- If Max Fanout is attached to a specific net, XST does not perform logic replication. Putting Max Fanout on the net may prevent XST from having better timing optimization.

For example, suppose that the critical path goes through the net, which actual fanout is 80 and set Max Fanout value to 100. If Max Fanout is specified in ISE, XST may replicate it, trying to improve timing. If Max Fanout is attached to the net itself, XST does not perform logic replication.

MAX_FANOUT for MAP

The MAX_FANOUT constraint can also drive MAP to limit fanout by duplicating registers and/or gates. MAP's register duplication option (-register_duplication) must be enabled and MAX_FANOUT constraints must be applied locally to nets for this to occur. When used during MAP, only the value of "REDUCE" is accepted. When MAX_FANOUT = "REDUCE", MAP will limit fanout if it determines that it can provide an improvement in performance with out causing problems in fitting the design. Review the physical synthesis report (*PSR) generated by MAP to review whether or not MAX_FANOUT = "REDUCE" caused fanout reduction to actually occur.

MAX_FANOUT Architecture Support

Max Fanout applies to all FPGA devices. Max Fanout does not apply to CPLD devices.

MAX_FANOUT Applicable Elements

When the value is an integer, Max Fanout applies globally, or to a VHDL entity, a Verilog module, or signal.

When the value is "REDUCE", Max Fanout only applies to a signal.

MAX_FANOUT Propagation Rules

Max Fanout applies to the entity, module, or signal to which it is attached.

MAX_FANOUT Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax

Before using Max Fanout, declare it with the following syntax:

```
attribute max_fanout: string;
```

After declaring Max Fanout, specify the VHDL constraint:

attribute max_fanout of {*signal_name* | *entity_name*}**:** {**signal** | **entity**} **is "***integer***";**

Verilog Syntax

Place Max Fanout immediately before the module or signal declaration:

(* max_fanout = "*integer***" *)**

XCF Syntax One

MODEL "*entity_name*" **max_fanout=***integer*;

XCF Syntax Two

BEGIN MODEL "*entity_name*"

NET "*signal_name*" **max_fanout=***integer*;

END;

XST Command Line Syntax

Define Max Fanout globally with the **-max_fanout** command line option of the **run** command:

-max_fanout *integer*

ISE Syntax

Define globally in ISE in **Process > Properties > Xilinx-Specific Options > Max Fanout**.

UCF Syntax

When used in conjunction with MAP's Register Duplication Option, specify the MAX_FANOUT constraint in your UCF as follows:

NET "*signal_name*" **max_fanout=REDUCE;**

MAXDELAY (Maximum Delay)

The MAXDELAY constraint defines the maximum allowable delay on a net.

MAXDELAY Architecture Support

This constraint applies to FPGA devices only.

MAXDELAY Applicable Elements

Nets

MAXDELAY Propagation Rules

Applies to the net to which it is attached

MAXDELAY Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net
- Attribute Name: MAXDELAY
- Attribute Values: *value units*

  value is the numerical time delay

  units are micro, ms, ns, ps

VHDL Syntax

Declare the VHDL constraint as follows:

attribute maxdelay: string;

Specify the VHDL constraint as follows:

attribute maxdelay of *signal_name*: **signal is "***value* [*units*]**";**

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.

Verilog Syntax

Place this attribute immediately before the module declaration or instantiation.

Specify the Verilog constraint as follows:

(*MAXDELAY = "*value** [*units*]**" *)**

value is any 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.

UCF and NCF Syntax

NET "*net_name***" MAXDELAY=***value units***;**

value is the numerical time delay.

units are micro, ns, ms, ps.

The following statement assigns a maximum delay of 10 nanoseconds to the net $SIG_4.

```
NET "$1I3245/$SIG_4" MAXDELAY=10 ns;
```

PCF Syntax

item **MAXDELAY =** *maxvalue* [**PRIORITY** *integer*];

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

Constraints Editor Syntax

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Exceptions, which is under Timing Constraints, in the Constraint Type list box, double-click Nets to access a dialog box.

FPGA Editor Syntax

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.

MAXPT (Maximum Product Terms)

MAXPT is an advanced 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 Architecture Support

This constraint applies to CPLD devices only.

MAXPT Applicable Elements

Signals

MAXPT Propagation Rules

Applies to the signal to which it is attached

MAXPT Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

VHDL Syntax

Declare the VHDL constraint as follows:

```
attribute maxpt:  integer;
```

Specify the VHDL constraint as follows:

attribute maxpt of *signal_name* **: signal is "***integer***";**

integer is any positive integer

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* MAXPT = "*integer***" *)**

integer is any positive integer

For more information on basic Verilog syntax, see Verilog.

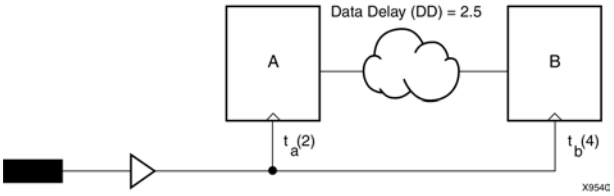
UCF and NCF Syntax

Net "*signal_name***" maxpt=***integer***;**

MAXSKEW (Maximum Skew)

MAXSKEW is a timing constraint used to control the maximum amount of skew on a net. This constraint is commonly used to control the skew of local clocks, or clocks that are not on the global clock network. This constraint is not necessary, or recommended, for global clock networks. Skew is defined as the difference between the delays of all loads driven by the net. Because the constraint identifies all loads driven by the net, skew may be reported between loads which have no logical connection.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.

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 Architecture Support

This constraint applies to FPGA devices only.

MAXSKEW Applicable Elements

Nets

MAXSKEW Propagation Rules

Applies to the net to which it is attached

MAXSKEW Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net
- Attribute Name: MAXSKEW
- Attribute Values: *allowable_skew* **[***units***]**

  allowable_skew is the timing requirement

  units are ms, micro, ns, or ps. The default is ns.

VHDL Syntax

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*]**";**

allowable_skew is the timing requirement

units are ms, micro, ns, or ps. The default is ns.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* MAXSKEW = "*allowable_skew* [*units*] **" *)**

allowable_skew is the timing requirement

units are ms, micro, ns, or ps. The default is ns.

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

NET " *net_name***" MAXSKEW=***allowable_skew* [*units*]**;**

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;
```

Constraints Editor Syntax

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Exceptions, which is under Timing Constraints, in the Constraint Type list box, double-click Nets to access a dialog box.

FPGA Editor Syntax

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.

MIODELAY_GROUP (MIODELAY Group)

MIODELAY_GROUP is a design implementation constraint that combines two or more IODELAY_GROUPs into a single Master IODELAY Group to enable automatic replication and placement of IDELAYCTRL(s) in a design.

MIODELAY_GROUP Architecture Support

The MIODELAY_GROUP constraint is applicable for the Virtex®-4 and Virtex-5 architectures. For Virtex-4, this constraint is only supported when using the Timing Driven Pack and Placement Option in MAP.

MIODELAY_GROUPApplicable Elements

MIODELAY_GROUP is applied to two or more defined IODELAY_GROUPs.

MIODELAY_GROUP Propagation Rules

MIODELAY_GROUP is a constraint that is applied to an existing IODELAY_GROUP. The MIODELAY_GROUP will be propagated to all of the design elements that belonged to the original IODELAY_GROUP(s). It is illegal to attach MIODELAY_GROUP to a net, signal, or pin.

MIODELAY_GROUP Syntax

The basic UCF syntax for defining an IODELAY_GROUP follows:

MIODELAY_GROUP "*master_group_name***" =** *iodelay_group1 iodelay_group2 ...* **;**

where

- *master_group_name* represents the master group being defined. It contains all of the elements in *iodelay_group1* and *iodelay_group2*
- *iodelay_group1* and *iodelay_group2* are predefined IODELAY groups (IODELAY_GROUP)

NODELAY (No Delay)

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®-3, Spartan-3A, Spartan-3E devices.

NODELAY can be attached to the I/O symbols and the special function access symbols TDI, TMS, and TCK.

NODELAY Architecture Support

Spartan-3, Spartan-3A, and Spartan-3E devices are supported.

IOBDELAY=NONE, which works for all FPGA devices, is the preferred method of applying this constraint. For more information see IOBDELAY (Input Output Block Delay).

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 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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: NODELAY
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

BEGIN MODEL "*entity_name* **"**

NET "*signal_name***" nodelay=true;**

INST "*instance_name***" nodelay=true;**

END;

NOREDUCE (No Reduce)

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 Architecture Support

This constraint applies to CPLD devices only.

NOREDUCE Applicable Elements

Any net

NOREDUCE Propagation Rules

NOREDUCE is a net or signal constraint. Any attachment to a design element is illegal.

NOREDUCE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net
- Attribute Name: NOREDUCE
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

```
BEGIN MODEL "entity_name"
NET "signal_name" noreduce={true|false};
END;
```

OFFSET IN (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.

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, MMCM, 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 Architecture Support

This constraint applies to all FPGA and CPLD devices.

OFFSET IN Applicable Elements

- Global
- Net-Specific
- Pad Time Group

OFFSET IN Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

While UCF examples are provided, the recommended method of specifying the OFFSET IN constraint is using the Constraint Editor software tool.

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 a 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.
- [{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.

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:

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:

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:

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

- Attach to a specific net
- Attribute Name: OFFSET
- Attribute Values: IN|OUT *offset_time* BEFORE|AFTER *clk_pad_netname*

XCF Syntax:

The XCF syntax is the same as the UCF syntax. However, the XCF syntax only supports OFFSET IN BEFORE method.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

Constraints Editor Syntax

From the ISE® Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Timing Constraints in the Constraint Type list box, double-click Inputs to access a dialog box.

OFFSET OUT (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 output 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, MMCM, 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 Architecture Support

This constraint applies to all FPGA and CPLD devices.

OFFSET OUT Applicable Elements

- Global
- Nets
- Time groups

OFFSET OUT Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

While UCF examples are provided, the recommended method of specifying the OFFSET OUT constraint is using the Constraint Editor software tool.

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.

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:

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:

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

- Attach to a specific net
- Attribute Name: OFFSET
- Attribute Values: OUT *offset_time* BEFORE|AFTER *clk_pad_netname*

XCF Syntax:

The XCF syntax is the same as the UCF syntax. However, the XCF syntax only supports OFFSET OUT AFTER method.

Constraints Editor Syntax

From the ISE® Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Timing Constraints in the Constraint Type list box, double-click Outputs to access a dialog box.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

OPEN_DRAIN (Open Drain)

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 Architecture Support

This constraint applies to CoolRunner-II devices only.

OPEN_DRAIN Applicable Elements

- Output pads
- Pad nets

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to an output pad net
- Attribute Name: OPEN_DRAIN
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

NET "*mysignal***" OPEN_DRAIN;**

XCF Syntax

BEGIN MODEL "*entity_name***"**

NET "*signal_name***" OPEN_DRAIN=true;**

END;

OPT_EFFORT (Optimizer Effort)

OPT_EFFORT is a basic placement and routing constraint. It defines an effort level used by the optimizer.

OPT_EFFORT Architecture Support

This constraint applies to FPGA devices only.

OPT_EFFORT Applicable Elements

Any macro or hierarchy level

OPT_EFFORT Propagation Rules

OPT_EFFORT is a macro, entity, module constraint. Any attachment to a net or signal is illegal.

OPT_EFFORT Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a macro
- Attribute Name: OPT_EFFORT
- Attribute Values: Default (Low), Lowest, Low, Normal, High, Highest

UCF and NCF Syntax

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

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 Standard.

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 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 Architecture Support

This constraint applies to FPGA devices only.

OPTIMIZE Applicable Elements

Any macro, entity, module or hierarchy level

OPTIMIZE Propagation Rules

Applies to the macro, entity, or module to which it is attached

OPTIMIZE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a design element
- Attribute Name: OPTIMIZE
- Attribute Values: AREA, SPEED, BALANCE, OFF

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify OPTIMIZE as follows:

```
(* OPTIMIZE = "{AREA|SPEED|BALANCE|OFF}" *)
```

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

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)

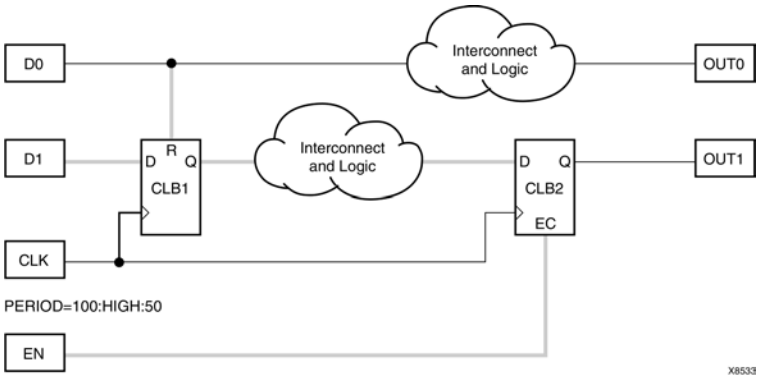
PERIOD is a basic timing constraint and synthesis constraint. A clock period specification checks timing between all synchronous elements within the clock domain as defined in the destination element group. The group may contain paths that pass between clock domains if the clocks are defined as a function of one or the other.

Derived period constraints are defined in terms of the same units as their reference constraint.

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).

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 that apply when using TNM and TNM_NET with the PERIOD constraint for DLLs, DCMs, PLLs, and MMCMs are discussed below.

PERIOD Architecture Support

This constraint applies to FPGA devices only.

PERIOD Applicable Elements

Nets that feed forward to drive flip-flop clock pins

PERIOD Propagation Rules

Applies to the signal to which it is attached

PERIOD Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

TIMESPEC PERIOD Method (primary method, recommended)

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

TIMESPEC "TS*identifier***"=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

```
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;
```

NET PERIOD Method (secondary method, not recommended)

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

PERIOD = *period* {**HIGH|LOW**} [*high_or_low_time*] **INPUT_JITTER** *value***;**

UCF Syntax

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. Xilinx® recommends using the same HIGH/LOW keyword on the derived PERIOD constraints as the master PERIOD constraint. If the master PERIOD constraint has the HIGH keyword or is the default, Xilinx recommends using the same HIGH keyword on the derived PERIOD constraints. To define the relationship in the case of a derived clock, use the following syntax:

UCF Syntax

TIMESPEC "TS*identifier***"=PERIOD "***timegroup_name***" "TS***identifier***"** [* 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;
```

Schematic Syntax

- 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

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.

Verilog Syntax

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.

UCF and NCF Syntax

TIMESPEC PERIOD Method (primary method, recommended)

UCF Syntax

TIMESPEC "TS*identifier***"=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, PLL, or MMCM component, new PERIOD specifications may be created at the DLL/DCM/PLL/MMCM 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/MMCM 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/MMCM output net to tag all synchronous elements controlled by that clock signal. The new groups and specifications are shown in the timing analysis reports.

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

From the ISE® Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Timing Constraints in the Constraint Type list box, double-click Clock Domains to access a dialog box.

XCF Syntax

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

"TS*identifier***"=PERIOD** *perioditem 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*

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

FPGA Editor Syntax

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.

PERIOD Specifications on CLKDLLs, DCMs, PLLs, and MMCMs

When a TNM or TNM_NET property traces into an input pin on a DLL, DCM, PLL, or MMCM, 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, PLL, or MMCM, the TNM group and its usage are examined. The TNM is pushed through the CLKDLL, DCM, PLL, or MMCM (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/MMCM, 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, PLL, or MMCM 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, PLL, or MMCM, 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.

New PERIOD Specifications

Output Pin	Period Value	Phase Shift	Duty Cycle
CLK0	TS_CLKIN * 1	none	Copied from TS_CLKIN if DUTY_CYCLE_CORRECTION is FALSE. Otherwise, 50%

Output Pin	Period Value	Phase Shift	Duty Cycle
CLK90	TS_CLKIN * 1	PHASE + (*clk0_period* * 1/4)	Copied from TS_CLKIN if DUTY_CYCLE_CORRECTION is FALSE. Otherwise, 50%
CLK180	TS_CLKIN * 1	PHASE + (*clk0_period* * 1/2)	Copied from TS_CLKIN if DUTY_CYCLE_CORRECTION is FALSE. Otherwise, 50%
CLK270	TS_CLKIN * 1	PHASE + (*clk0_period* * 3/4)	Copied from TS_CLKIN if DUTY_CYCLE_CORRECTION is FALSE. Otherwise, 50%
CLK2X	TS_CLKIN / 2	none	50%
CLK2X180	TS_CLKIN / 2	PHASE + (*clk2X_period* * 1/2)	50%
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		none	
CLKFX180	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).	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.

PIN (Pin)

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 Architecture Support

This constraint applies to FPGA devices only.

PIN Applicable Elements

Nets

PIN Propagation Rules

Not applicable.

PIN Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

PIN " *module.pin* **" LOC=***location* **;**

PIN mod.pin **TIG;**

POST_CRC (Post_CRC)

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 the Spartan-3A device, 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 PlanAhead™, 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 Architecture Support

The POST_CRC constraint applies to Virtex®-5, Virtex-6, Spartan®-3A, and Spartan-6 devices.

POST_CRC Applicable Elements

This constraint relates to the entire device and is not specified on any particular design element.

POST_CRC Propagation Rules

This constraint applies to the entire design/device.

POST_CRC Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

```
CONFIG POST_CRC = {ENABLE|DISABLE|ONESHOT};
```

PCF Syntax

```
CONFIG POST_CRC = {ENABLE|DISABLE|ONESHOT};
```

POST_CRC_ACTION (Post CRC Action)

Spartan®-3A, Spartan-6, and Virtex®-6 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 (Default for Spartan-6).
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 for Virtex-6).
CORRECT_AND_CONTINUE	If a CRC mismatch is detected by the CRC comparison, it is corrected and continues reading back the bitstream, computing the comparison CRC, and making the comparison against the pre-computed CRC.
CORRECT_AND_HALT	If a CRC mismatch is detected, it is corrected and ceases reading back the bitstream, computing the comparison CRC, and making the comparison against the pre-computed CRC.

POST_CRC_ACTION Architecture Support

This constraint applies to Spartan-3A, Spartan-6, and Virtex-6 devices.

POST_CRC_ACTION Applicable Elements

This constraint relates to the entire device and is not specified on any particular design element.

POST_CRC_ACTION Propagation Rules

This constraint applies to the entire design/device.

POST_CRC_ACTION Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

```
CONFIG POST_CRC_ACTION = [HALT|CONTINUE];
```

PCF Syntax

```
CONFIG POST_CRC_ACTION = [HALT|CONTINUE];
```

POST_CRC_FREQ (Post CRC Frequency)

Spartan®-3A, Spartan-6, and Virtex®-6 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.

For Spartan-3A, 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.

For Spartan-6, the frequency range represented by these 10 bits are from 1 to 100 MHz, and the steps are 1, 2, 4, 6, 10, 12, 16, 22, 26, 33, 40, 50, and 66 MHz. The default value is 1 MHz.

For Virtex-6, the frequency range represented by these 10 bits are from 1 to 50 MHz, and the steps are 1, 2, 3, 6, 13, 25, and 50 MHz. The default value is 1 MHz.

POST_CRC_FREQ Architecture Support

This constraint applies to Spartan-3A, Spartan-6, and Virtex-6 devices.

POST_CRC_FREQ Applicable Elements

This constraint relates to the entire device and is not specified on any particular design element.

POST_CRC_FREQ Propagation Rules

This constraint applies to the entire design/device.

POST_CRC_FREQ Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

```
CONFIG POST_CRC_FREQ = [1|3|6|7|8|10|12|13|17|22|25|27|33|44|50|100];
```

PCF Syntax

```
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)

Virtex®-5 devices support a configuration logic CRC error detection mode called POST_CRC. In this mode, 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 INIT_B pin is enabled as an output for the SEU (Single Event Upset) error signal. 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 Architecture Support

This constraint applies to Virtex-5 devices only.

POST_CRC_SIGNAL Applicable Elements

This constraint relates to the entire device and is not specified on any particular design element.

POST_CRC_SIGNAL Propagation Rules

This constraint applies to the entire design/device.

POST_CRC_SIGNAL Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

```
CONFIG POST_CRC_SIGNAL = [FRAME_ECC_ONLY|INIT_AND_FRAME_ECC];
```

PCF Syntax

```
CONFIG POST_CRC_SIGNAL = [FRAME_ECC_ONLY|INIT_AND_FRAME_ECC];
```

PRIORITY (Priority)

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 Architecture Support

This constraint applies to all FPGA and CPLD devices.

PRIORITY Applicable Elements

TIMESPECs

PRIORITY Propagation Rules

Not applicable

PRIORITY Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

Defines the priority of a timing constraint using the following syntax.

normal_timespec_syntax **PRIORITY** *integer*;

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

Same as UCF

PROHIBIT (Prohibit)

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.

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®-3, Spartan-3A, Spartan-3E, 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-3, Spartan-3A, Spartan-3E, 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-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices
Slice	SLICE_X22Y3	Slice location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices

Element Type	Location Specification	Meaning
block RAM	RAMB16_X2Y56	Block RAM location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices
Multiplier	MULT18X18_X55Y82	Multiplier location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices
Global Clock	BUFGMUX0P	Global clock buffer location for Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices
Digital Clock Manager (DCM)	DCM_X[A]Y[B]	Digital Clock Manager for Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices
Phase Lock Loop (PLL)	PLL_X[A]Y[B]	Phase Lock Loop for Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices
Mixed-Mode Clock Manager (MMCM)	MMCM_X[A]Y[B]	Mixed-Mode Clock Manager for Virtex-6

You can use the wildcard character (*) to replace a single location with a range as shown in the following example.

SLICE_X*Y5	Any slice of an FPGA device whose Y-coordinate is 5

The following are *not* supported:

- Dot extensions on ranges. For example, LOC=SLICE_X3Y5:SLICE_X5Y7.G.
- The wildcard character for Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 global buffers 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 Architecture Support

This constraint applies to all FPGA and CPLD devices.

PROHIBIT Applicable Elements

Sites

PROHIBIT Propagation Rules

It is illegal to attach PROHIBIT to a net, signal, entity, module, or macro.

PROHIBIT Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

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 below. 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;
```

The following statement prohibits use of the slice at the SLICE_X6Y8 site.

```
CONFIG PROHIBIT=SLICE_X6Y8;
```

PCF Syntax

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)

PlanAhead Syntax

For more information about creating area group constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PACE Syntax

PACE (Pin Assignments Editor) can be used to set PROHIBIT. For more information, see the Prohibit Mode command section in the PACE help.

Note PACE is supported only for CPLDs.

FPGA Editor Syntax

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 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 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 Propagation Rules

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

PULLDOWN Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a pad net
- Attribute Name: PULLDOWN
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

```
(* PULLDOWN = “{YES|NO|TRUE|FALSE}” *)
```

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

```
BEGIN MODEL "entity_name"
NET "signal_name" pulldown=true;
END;
```

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PULLUP (Pullup)

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 Architecture Support

The PULLUP constraint applies to all FPGA devices and the CoolRunner XPLA3 and CoolRunner-II CPLD devices.

PULLUP Applicable Elements

- Input
- Tristate outputs
- Bidirectional pad nets

PULLUP Propagation Rules

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

PULLUP Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a pad net
- Attribute Name: PULLUP
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

```
(* PULLUP = “{YES|NO|TRUE|FALSE}” *)
```

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

BEGIN MODEL “*entity_name*”

NET "*signal_name*" **pullup=true;**

```
END;
```

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PWR_MODE (Power Mode)

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 Architecture Support

This constraint applies to XC9500 devices only:

PWR_MODE Applicable Elements

- Nets
- Any instance

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net or an instance
- Attribute Name: PWR_MODE
- Attribute Values: LOW, STD

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* PWR_MODE = "{**LOW**|**STD**} *)

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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

BEGIN MODEL "*entity_name***"**

NET "*signal_name***" PWR_MODE=**{**LOW**|**STD**}**;**

INST "*instance_name***" PWR_MODE=**{**LOW**|**STD**}**;**

END;

REG (Registers)

REG is a basic fitter constraint. It specifies how a register is to be implemented in the CPLD macrocell.

REG Architecture Support

This constraint applies to CPLD devices only.

REG Applicable Elements

Registers

REG Propagation Rules

When attached to a design element, REG propagates to all applicable elements in the hierarchy within the design element.

REG Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a flip-flop instance or macro containing flip-flops
- Attribute Name: REG
- Attribute Values: CE, TFF

VHDL Syntax

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 the UCF and NCF Syntax for this constraint.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* REG = {**CE**|**TFF**} ***)**

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

The basic UCF syntax is:

INST “*instance_name*” **REG =** {**CE** | **TFF**}**;**

- 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 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 macrocell.

```
INST “Q1” REG=CE;
```

XCF Syntax

BEGIN MODEL “*entity_name*”

NET "*signal_name*" **REG=**{**CE**|**TFF**}**;**

END;

RLOC (Relative Location)

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 Spartan®-3A, Spartan-3E, Virtex®-4 and Virtex-5 devices, the RLOC constraint is specified using the slice-based XY coordinate system.

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 the Set Modifiers section below. 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.

- Guidelines for Specifying Relative Locations
- RLOC Sets

Although RLOC constraints control the relative placement of logic blocks, they do not guarantee that the same routing resources are used to connect the logic blocks from implementation to implementation. In order to control the routing used, please refer to DIRECTED_ROUTING constraints.

RLOC Architecture Support

This 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 Guides for details. Also for more information, see the device data sheet.

- Registers
- ROM
- RAMS, RAMD
- 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.
- LUTs, MUXF5, MUXF6, MUXCY, XORCY, MULT_AND, SRL16, SRL16E, MUXF7 (Spartan-3, Spartan-3A, Spartan-3E devices only)
- MUXF8 (all FPGA devices only)
- Block RAMs
- Multipliers
- DSP48

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.

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.

RLOC Syntax

The RLOC constraint is specified using the slice-based XY coordinate system.

RLOC=X*m***Y** *n*

m is an integer representing the X coordinate

n is an integer representing the Y coordinate

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** 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.

- RLOC_ORIGIN (Relative Location Origin) sets the exact die location of the set members. This constraint lets you change the RLOC values into absolute LOC constraints that respect the structure of the set.

  The design resolution program, NGCBuild, 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.

- RLOC_RANGE (Relative Location Range) limits 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.

  You cannot use this constraint on sets that include BUFT symbols.

- USE_RLOC turns the RLOC constraints on and off for a specific element or section of a set. USE_RLOC can be either TRUE or FALSE.

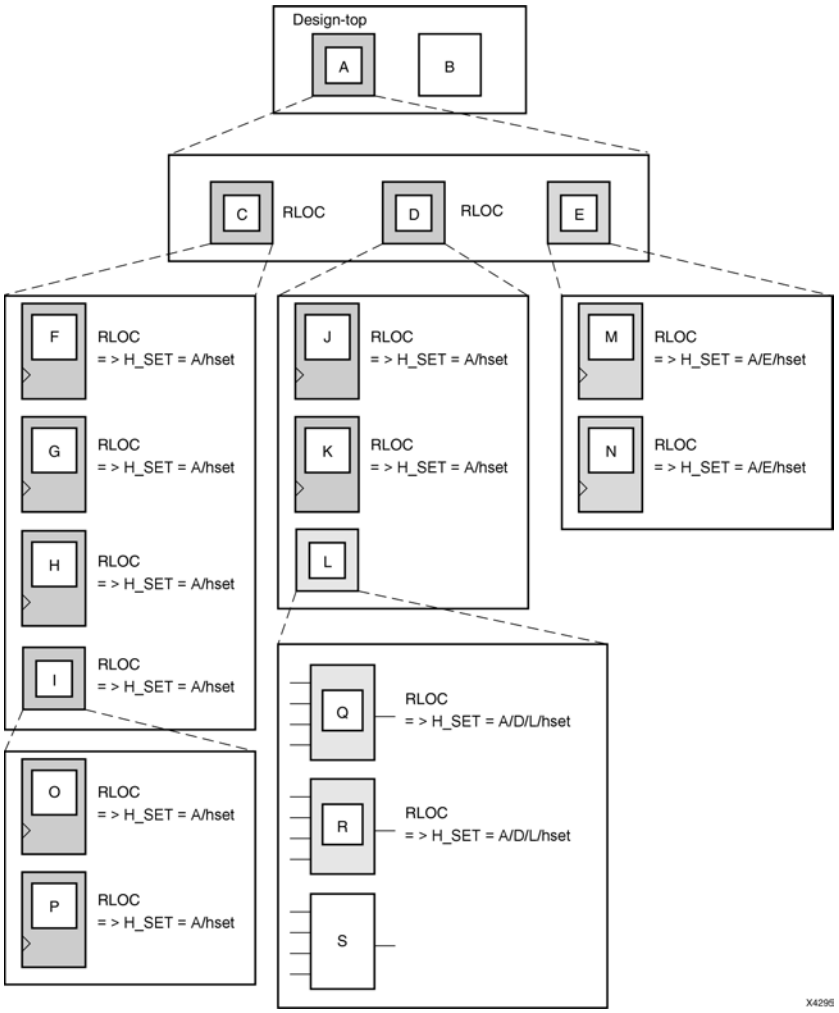
  The application of the USE_RLOC constraint is strictly based on hierarchy. A USE_RLOC constraint attached to an element applies to all its underlying elements that are members of the same set. If it is attached to a symbol that defines the start of a set, the constraint is applied to all the underlying member elements, which represent the entire set.

  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.

  You can attach the USE_RLOC constraint directly to a primitive symbol so that it affects only that symbol.

Linking Sets

Set Linkage



This example shows the process of linking together elements through the design hierarchy. 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.

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 this figure, 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. 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."

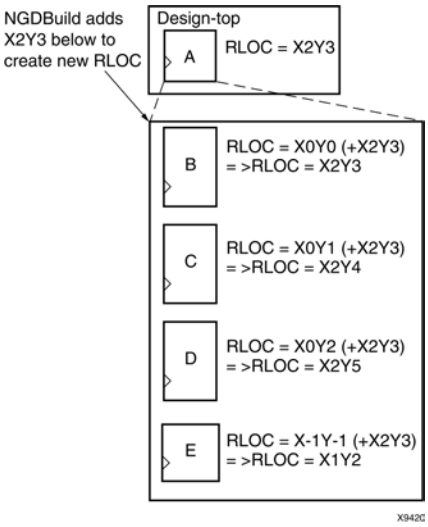
Modifying Sets

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 the Three H_SET Sets example, 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.

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.

The following sections describe the effect of the hierarchy on set modification.

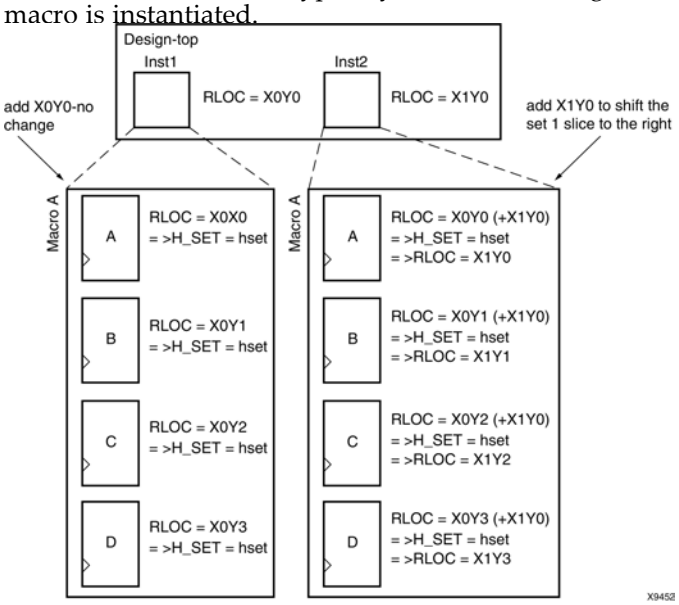
Adding RLOC Values Down the Hierarchy Example (Slice-based XY Designations)



This example 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.

Modifying RLOC Values of Same Macro and Linking Together as One Set

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.

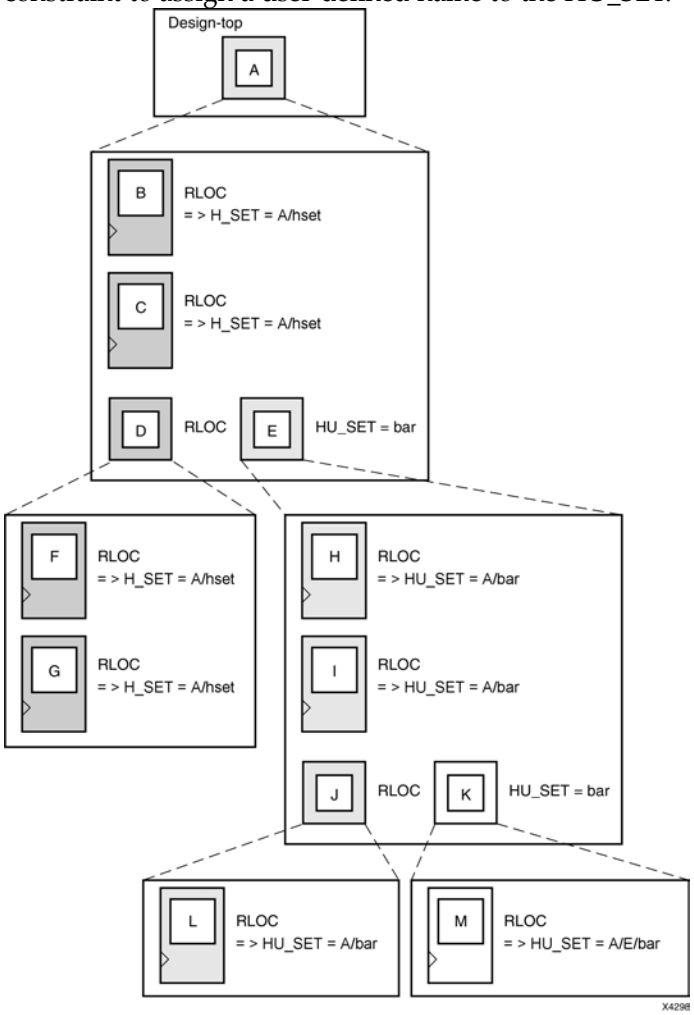


This example is a variation of the previous example. The RLOC constraint on Inst1 and Inst2 now link all the objects in one H_SET set.

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.

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.



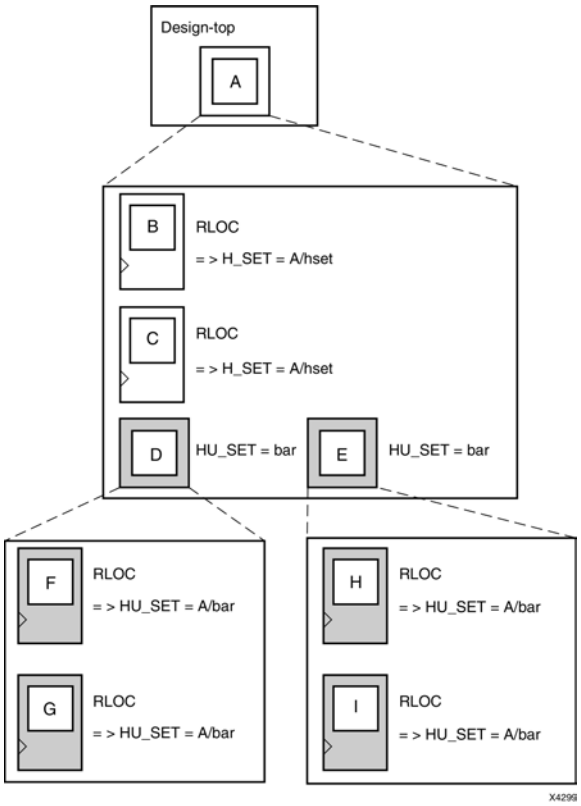
This example demonstrates how HU_SET constraints designate elements as set members, break links between elements tagged with RLOC constraints in the hierarchy to separate them from H_SET sets, and generate names as identifiers of these sets.

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.

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.

Linking Two HU_SET Sets

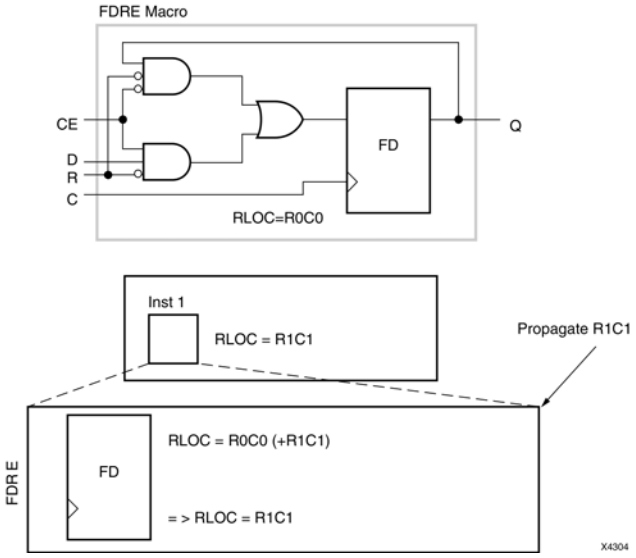


This example shows how HU_SET constraints link elements in the same node together by naming them with the same identifier. Because of the same name, "bar," on two elements, D and E, the elements tagged with RLOC constraints below D and E become part of the same HU_SET.

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. 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 Xilinx flip-flop macro. The mapper adds the RLOC value that you specified to the underlying primitive so that it has the desired value.



In this example, 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.

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.

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.

RLOC Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to an instance
- Attribute Name: RLOC
- Attribute Values: See Syntax in this chapter.

VHDL Syntax

Declare the VHDL constraint as follows:

```
attribute rloc:  string;
```

Specify the VHDL constraint as follows for all FPGA devices:

attribute rloc of {*component_name*|*entity_name*| *label_name*}**:** {**component** |**entity**|**label**} **is** **“**[*element*]**X** *m***Y***n*[**.***extension*]**”;**

For descriptions of valid values, see Guidelines for Specifying Relative Locations.

For more information on basic VHDL syntax, see VHDL.

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.

Note User needs to create the “itoa” function.

```
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

Place this constraint immediately before an instantiation:

Specify as follows for all FPGA devices:

(* RLOC = “[*element*]**X***m***Y** *n*[**.***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

For all FPGA 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

For Virtex-4 and Virtex-5 devices:

BEGIN MODEL “*entity_name* **”**

INST "*instance_name* **" rloc=**[*element*]*XmYn* [**.***extension*] **;**

END;

PlanAhead Syntax

For more information about creating area group constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also seePlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

Guidelines for Specifying Relative Locations

The slice-based coordinate system for assigning elements to relative location uses the general syntax

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 and higher and Virtex®-4 and higher 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.

Different RLOC Specifications for Four Flip-Flop Primitives

This figure demonstrates the use of RLOC constraints. In (a), 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). 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.

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 four flip-flops in Different RLOC Specifications for Four Flip-Flop Primitives 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

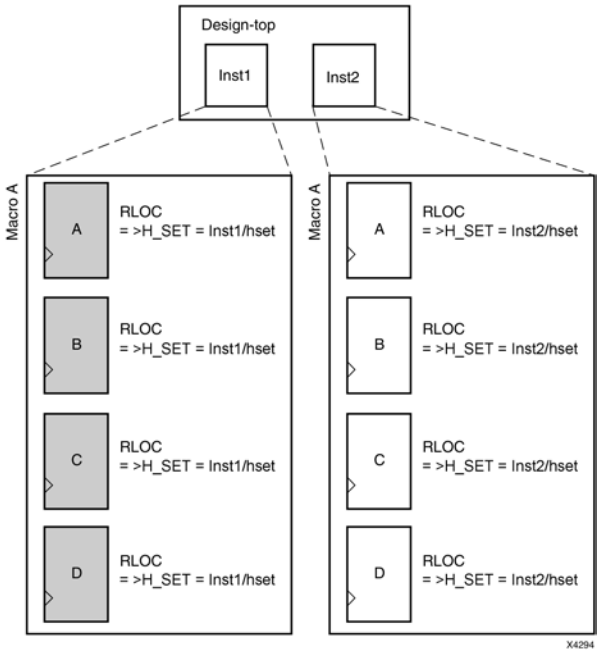
This user-defined name is the base name of the HU_SET set. Like the H_SET set, in which the base name of "h_set" is prefixed by the hierarchical name of the lowest common ancestor of the set elements, the user-defined base name of an HU_SET set is prefixed by the hierarchical name of the lowest common 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.

Macro A Instantiated Twice



Note In this figure 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.

This figure 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 and higher and Virtex®-4 and higher devices 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. 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.

This figure 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.

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_ORIGIN (Relative Location Origin)

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 Architecture Support

This 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.

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.

RLOC_ORIGIN Syntax

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.

set_name **RLOC_ORIGIN=X***m* **Y***n*

- *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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to an instance that is a member of a set
- Attribute Name: RLOC_ORIGIN
- Attribute Values: See the UCF and NCF Syntax for this constraint.

VHDL Syntax

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 Spartan®-3, Spartan-3A, Spartan-3E, Virtex®-4 and Virtex-5 devices, *value* is **X** *m***Y***n*.

For a description of valid values, see the UCF and NCF Syntax below.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* RLOC_ORIGIN = “ *value***” *)**

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices, *value* is **X** *m***Y***n*.

For a description of valid values, see the UCF and NCF Syntax below.

For more information on basic Verilog syntax, see Verilog .

UCF and NCF Syntax for Architectures Using Slice-Based XY Coordinates

This section applies to Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices.

RLOC_ORIGIN=X *m***Y***n*

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;

PlanAhead Syntax

For more information about creating area group constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

RLOC_RANGE (Relative Location Range)

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 Architecture Support

This constraint applies to FPGA devices only.

RLOC_RANGE Applicable Elements

Instances or macros that are members of sets

RLOC_RANGE Propagation Rules

RLOC_RANGE is a macro constraint and any attachment to a net is illegal.

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.

RLOC_RANGE Syntax

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.

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.

RLOC_RANGE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to an instance that is a member of a set
- Attribute Name: RLOC_RANGE
- Attribute Values: See the UCF and NCF Syntax below.

VHDL Syntax

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 Spartan®-3, Spartan-3A, Spartan-3E, Virtex®-4 and Virtex-5 devices *value* is **X***m1***Y***n1*:**X***m2***Y***n2*.

For a description of valid values, see the UCF and NCF Syntax below.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* RLOC_RANGE = "*value***" *)**

For Spartan-3, Spartan-3A, Spartan-3E, Virtex-4 and Virtex-5 devices, *value* is **X***m1***Y***n1*:**X***m2***Y***n2*.

For a description of valid values, see the UCF and NCF Syntax below.

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

This section is applicable Spartan-3 devices and up, and Virtex-4 devices and up.

RLOC_RANGE=X*m1***Y***n1***:X***m2***Y***n2*

the relative X values (*m1* and *m2*) and Y values (*n1* and *n2*) can be:

- positive integers (including zero)
- the wildcard (*) character

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

MODEL "*entity_name***" rloc_range=***value***;**

BEGIN MODEL "*entity_name***"**

INST "*instance_name***" rloc_range=***value***;**

END;

SAVE NET FLAG (Save Net Flag)

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 Architecture Support

This constraint applies to FPGA devices only.

SAVE NET FLAG Applicable Elements

- Nets
- Signals

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net or signal
- Attribute Name: S
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

The following statement specifies that the net or signal named $SIG_9 should not be removed.

```
NET $SIG_9 S;
```

XCF Syntax

BEGIN MODEL *entity_name*

NET "*signal_name***" s=true;**

```
END;
```

SCHMITT_TRIGGER (Schmitt Trigger)

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 Architecture Support

This constraint applies to CoolRunner™-II devices only.

SCHMITT_TRIGGER Applicable Elements

All input pads and pad nets

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net
- Attribute Name: SCHMITT_TRIGGER
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog .

UCF and NCF Syntax

```
NET "mysignal" SCHMITT_TRIGGER;
```

XCF Syntax

BEGIN MODEL "*entity_name* **"**

NET "*signal_name***" SCHMITT_TRIGGER=true;**

```
END;
```

SLEW (Slew)

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.

SLEW Architecture Support

This constraint applies to all FPGA and CPLD devices.

SLEW Applicable Elements

Output primitives, output pads, bidirectional pads.

You can also attach SLEW to the net connected to the pad component in a UCF file. NGCBuild transfers SLEW 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***" slew=**{**FAST**|**SLOW**};

SLEW Propagation Rules

The SLEW attribute should only be placed on a top-level output or bi-directional port.

SLEW Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

Specify a new attribute to an output port, or bi-directional port:

- Attribute Name: SLEW
- Attribute Values: FAST, SLOW, QUIETIO (Spartan-3A device only)

VHDL Syntax

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 device 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.

Verilog Syntax

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 device only)

Example:

```
module top (
(* SLEW="FAST" *) output FAST_OUT
);
```

For a more detailed discussion of the basic Verilog syntax, see Verilog.

UCF and NCF Syntax

Placed on output or bi-directional port:

NET "top_level_port_name" SLEW="*value***";**

value is SLOW, FAST, QUIETIO (Spartan-3A device only).

Example:

```
NET "FAST_OUT" SLEW="FAST";
```

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PACE Syntax

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.

Note PACE is supported only for CPLDs.

SLOW (Slow)

SLOW is a basic fitter constraint. It stipulates that the slew rate limited control should be enabled.

SLOW Architecture Support

This 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. NGCBuild 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 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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: SLOW
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

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

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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;
```

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

STEPPING (Stepping)

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 Architecture Support

- CoolRunner™-II CPLD architecture
- Spartan®-3A, Spartan-3E, Virtex®-4, and Virtex-5 FPGA architectures

STEPPING Applicable Elements

The STEPPING attribute is a global CONFIG constraint and is not attached to any instance or signal name.

STEPPING Propagation Rules

The CONFIG STEPPING constraints applies to an entire design.

STEPPING Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF Syntax

`CONFIG STEPPING="`*n*`";`**

n is the target stepping level (ES, SCD1, 1, 2, 3, ...)

For example:

```
CONFIG STEPPING="1";
```

SUSPEND (Suspend)

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 Architecture Support

This 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 Propagation Rules

The SUSPEND attribute should only be placed on a top-level output or bi-directional port.

SUSPEND Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

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

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***";**

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.

Verilog Syntax

Place the following attribute specification before the port declaration in the top-level Verilog code:

(* SUSPEND="*value***" *)**

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.

UCF and NCF Syntax

Placed on output or bi-directional port:

NET "*top_level_port_name***" SUSPEND="***value***";**

value is DRIVE_LAST_VALUE, 3STATE, 3STATE_PULLUP, 3STATE_PULLDOWN or 3STATE_KEEPER

Example:

```
NET "STATUS" SUSPEND="DRIVE_LAST_VALUE";
```

PACE Syntax

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)

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 on a global basis to all of the clocks within a design. It is combined with the INPUT_JITTER keyword on the PERIOD constraint, as well as any jitter or phase error in the clock network, to generate the Clock Uncertainty value that is shown in the timing report.

SYSTEM_JITTER Architecture Support

This constraint applies to FPGA devices only.

SYSTEM_JITTER Applicable Elements

Applies globally to the entire design

SYSTEM_JITTER Propagation Rules

Not applicable

SYSTEM_JITTER Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: SYSTEM_JITTER

VHDL Syntax

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";**

value is a numerical value. The default is ps.

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* SYSTEM_JITTER = "*value*** ps" *)**

value is a numerical value. The default is ps.

For more information on basic Verilog syntax, see Verilog .

UCF and NCF Syntax

The basic UCF syntax is:

SYSTEM_JITTER= *value* **ps;**

value is a numerical value. The default is ps.

XCF Syntax

MODEL "*entity_name* **" SYSTEM_JITTER =** *value* **ps;**

TEMPERATURE (Temperature)

TEMPERATURE is a timing constraint that allows the specification of the operating junction temperature. TEMPERATURE provides a means of using 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 Architecture Support

The following FPGAs are supported for TEMPERATURE:

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

TEMPERATURE Applicable Elements

Global

TEMPERATURE Propagation Rules

It is illegal to attach TEMPERATURE to a net.

TEMPERATURE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

TEMPERATURE=*value* [**C |F| K];**

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

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Timing Constraints in the Constraint Type list box, double-click Operating Conditions to access a dialog box.

TIG (Timing Ignore)

TIG (Timing IGnore) is a 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 timing analysis) 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 Architecture Support

This constraint applies to FPGA devices only.

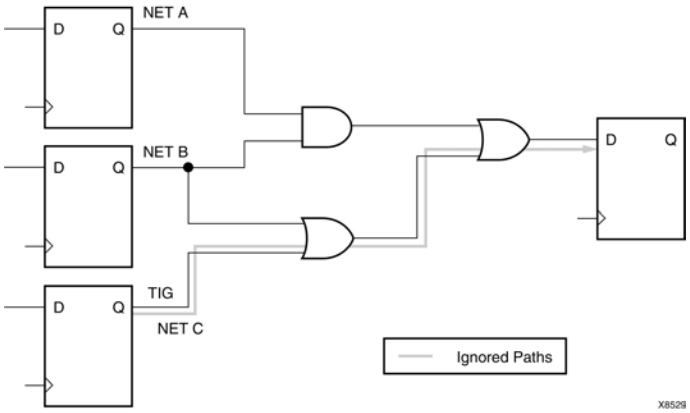
TIG Applicable Elements

- Nets
- Pins
- Instances

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.

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

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

- Attach to a net or pin.
- Attribute Name: TIG
- Attribute Values: *value*

UCF and NCF Syntax

The basic UCF syntax is:

`NET “` *net_name*`” TIG;`

`PIN “`*ff_inst*`.RST” TIG=TS_1;`

`INST “`*instance_name* `” TIG=TS_2;`

`TIG=TS` *identifier1 . . .* **`TS`** *identifiern*

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

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

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Exceptions, which is under Timing Constraints, in the Constraint Type list box, double-click Nets to access a dialog box.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PCF Syntax

item **TIG;**

item **TIG =;**

item **TIG = TS***identifier* **;**

item is one of the following:

- **PIN** *name*
- **PATH** *name*
- *path specification*
- **NET** *name*
- **TIMEGRP** *name*
- **BEL** *name*
- **COMP** *name*
- **MACRO** *name*

TIMEGRP (Timing Group)

TIMEGRP is constraint used to group design elements together for timing analysis. In addition to defining groups of design elements 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).

TIMEGRP Architecture Support

This constraint applies to all FPGA and CPLD devices.

TIMEGRP Applicable Elements

- Design elements
- Nets

TIMEGRP Propagation Rules

Applies to all elements or nets within the group

TIMEGRP Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

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 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 Two

A circular definition, as shown below, causes an error when you run your design through NGCBuild:

```
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 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 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 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 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

```
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.

UCF Syntax One

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 the UCF and NCF Syntax 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 Two

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 Three

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 Four

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 One

TIMESPEC "TS*identifier***"=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 Two

```
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.

UCF Syntax

Following are TIMEGRP UCF Syntax.

Example One

TIMEGRP **"***newgroup***"="***existing_grp1***" "***existimg_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

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

From the ISE® Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Group Constraints in the Constraint Type list box, double-click either Upstream Elements by Nets or By Clock Edge to access a dialog box.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

PCF Syntax

TIMEGRP *name***;**

TIMEGRP *name* **=** *list of elements***;**

TIMESPEC (Timing Specifications)

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 Architecture Support

This constraint applies to all FPGA and CPLD devices.

TIMESPEC Applicable Elements

TS identifiers

TIMESPEC Propagation Rules

Not applicable

TIMESPEC Syntax

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.

FROM-TO Syntax

Within TIMESPEC, you use the following UCF syntax to specify timing requirements between specific end points.

`TIMESPEC “TS`*identifier*`”=FROM “`***`source_group`***`” TO “`***`dest_group`***`” value units;`**

`TIMESPEC “TS`*identifier*`”=FROM “`***`source_group`***`” value units;`**

`TIMESPEC “TS`*identifier*`”=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 UCF and NCF Syntax section in the TNM_NET constraint, 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 Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

TIMESPEC Examples of FROM-TO TS Attributes

UCF and NCF Syntax -

```
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;
```

UCF Syntax

A TS attribute defines the allowable delay for paths in your design. The basic syntax for a TS attribute is:

TIMESPEC "TS*identifier*"**=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 "TS*identifier*"**=PERIOD "***timegroup_name*"** **"TS***identifier*"** [* or /] *factor* **PHASE** [**+** | **-**] *phase_value* [*units*]**;**

Constraints Editor Syntax

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."

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

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, 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.

Note A TNM based upon a PAD name that is associated with a Partition is not supported. A TNM based upon a net name within a Partition is supported.

TNM Architecture Support

This 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. NGCBuild 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 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.

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

Placing TNMs on Primitive Pins

You can place TNM on any component pin in the design if the design entry package allows placement of constraints onprimitive 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 syntax for the UCF file is:

PIN "*pin_name***" TNM="FLOPS";**

Placing TNMs on Primitive Symbols

You can group individual logic primitives explicitly by placing a constraint on each instance

The flip-flops tagged with TNM form a group called "'FLOPS." The untagged flip-flops are not part of the group. See the UCF syntax example.

Place only one TNM on each symbol, driver pin, or macro driver pin.

UCF Syntax

INST "*instance_name***" TNM=FLOPS;**

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 TNM parameter on nets or pins is allowed to have a qualifier. For example, 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=RAMS 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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

{**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 registered multiplier.
  - 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.

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 reserved words in the table below as *identifier*.

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

See the UCF and NCF Syntax for this constraint.

Constraints Editor Syntax

From the ISE® Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Group Constraints in the Constraint Type list box, double-click either By Instance/Hierarchy or By DCM Outputs to access a dialog box.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

TNM_NET (Timing Name Net)

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/PLL/MMCMs. For more information, see PERIOD Specifications on CLKDLLs, DCMs, PLLs, and MMCMs 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. NGCBuild never transfers a TNM_NET constraint from the attached net to an input pad, as it does with the TNM constraint.

TNM_NET Architecture Support

This constraint applies to FPGA devices only.

TNM_NET Applicable Elements

Nets

TNM_NET Rules

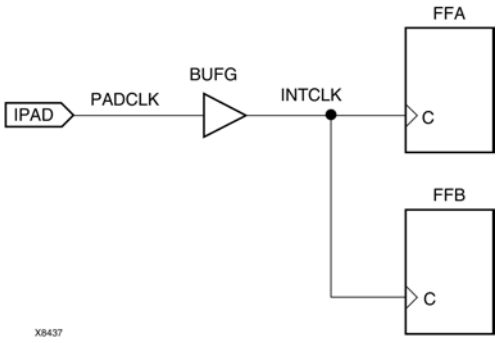
The following rules apply to TNM_NET:

- 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®-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

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net
- Attribute Name: TNM_NET
- Attribute Values: *identifier*

For a discussion of *identifier,* see the UCF and NCF Syntax for this constraint.

UCF and NCF Syntax

{**NET**|**INST**} **"***net_name***" TNM_NET=**[*predefined_group***:**]*identifier***;**

- *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 registered multiplier.
    - DSPS is used to group DSP elements like the Virtex-4 DSP48.
    - 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 reserved words shown in the TNM (Timing Name) constraint Reserved Words table as *identifier*.

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

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

From the ISE® Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Group Constraints in the Constraint Type list box, double-click Downstream Elements by Nets to access a dialog box.

PlanAhead Syntax

For more information about creating constraints using PlanAhead™, see the *Floorplanning the Design* section of the *PlanAhead Users Guide*.

Also see PlanAhead in this manual for information about the following:

- Defining placement constraints
- Assigning placement constraints
- Defining I/O pin configurations
- Floorplanning and placement constraints

TPSYNC (Timing Point Synchronization)

TPSYNC is a 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.

TPSYNC Architecture Support

This constraint applies to FPGA devices only.

TPSYNC Applicable Elements

- Nets
- Instances
- Pins

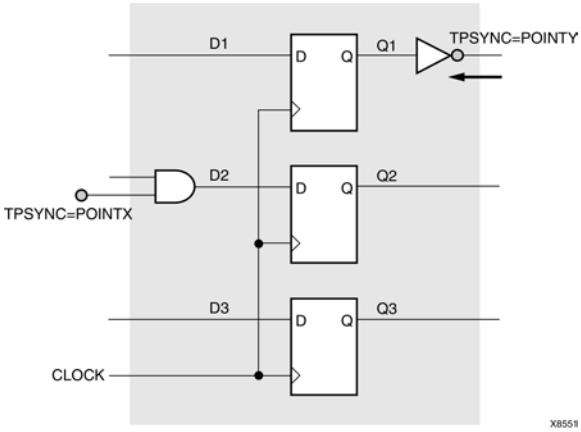
TPSYNC Propagation Rules

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.

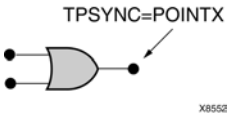
  In the following diagram, POINTY applies to the inverter.

TPSYNCs Attached to Macro Pins



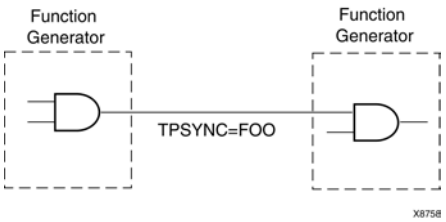
- The output pin of a primitive — the primitive's output is flagged as a potential source or destination for timing specifications.
- The input pin of a primitive — the primitive's input is flagged as a potential source or destination for timing specifications.
- An instance — the output of that element 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.

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.

Working with Two Gates



In this example, because of the TPSYNC definition, the two gates cannot be merged into a single function generator.

TPSYNC Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attached to a net, instance, or pin
- Attribute Name: TPSYNC
- Attribute Values: *identifier*

  identifier is a name that is used in timing specifications in the same way that groups are used

UCF and NCF Syntax

NET "*net_name*" **TPSYNC=***identifier*;

INST "*instance_name*" **TPSYNC=***identifier*;

PIN "*pin_name*" **TPSYNC=***identifier*;

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;
```

Constraints Editor Syntax

From the ISE® Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Group Constraints in the Constraint Type list box, double-click By Combinatorial Pinsto access a dialog box.

TPTHRU (Timing Thru Points)

TPTHRU is a 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 TIMESPEC (Timing Specifications) 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 TSidentifier (Timing Specification Identifier) constraint.

TPTHRU Architecture Support

This constraint applies to FPGA devices only.

TPTHRU Applicable Elements

- Nets
- Pins
- Instances

TPTHRU Propagation Rules

Not applicable

TPTHRU Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net, instance, or pin
- Attribute Name: TPTHRU
- Attribute Values: *identifier*

  For a discussion of *identifier,* see the UCF and NCF Syntax for this constraint.

UCF and NCF Syntax

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 "TS*identifier*"**=FROM "***source_group*" **THRU "***thru_point*" [**THRU "***thru_point*"] **TO "***dest_group*" *allowable_delay* [*units*];

TIMESPEC "TS*identifier*"**=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
```

Constraints Editor Syntax

From the ISE® Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Group Constraints in the Constraint Type list box, double-click By Through Points to access a dialog box.

PCF Syntax

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.

TSidentifier (Timing Specification Identifier)

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 Architecture Support

This constraint applies to all FPGA and CPLD devices.

TSidentifier Applicable Elements

TIMESPEC keywords

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

Defining a Maximum Allowable Delay

TIMESPEC "TS*identifier***"=FROM "***source_group***" TO "***dest_group***"** *allowable_delay* [*units*]**;**

Defining Intermediate Points (UCF)

TIMESPEC "TS*identifier***"=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 "TS*identifier***"=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 "TS*identifier***"=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 "TS*identifier***"=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 "TS*identifier***"=FROM "***source_group***" TO "***dest_group***" TIG;**

or

TIMESPEC "TS*identifier***"=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

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints.

Clock period constraints are entered using the Clock Domains entry. Input setup time is entered using the Inputs entry. Clock-to-output delay is entered using the Outputs entry. Pad-to-pad delays are entered using the **Exceptions** > **Paths** category.

PCF Syntax

The same as the UCF syntax without the TIMESPEC keyword.

FPGA Editor Syntax

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)

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 RLOC Sets.

U_SET Architecture Support

This 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 Guides* for details. Also for more information, see the device data sheet.

- Registers
- FMAP
- Macro Instance
- ROM
- RAMS, RAMD
- BUFT
- MULT18X18S
- RAMB4_Sm_Sn, RAMB4_Sn
- RAMB16_*Sm_Sn, RAMB16_Sn*
- RAMB16
- DSP48

U_SET Propagation Rules

U_SET is a macro constraint and any attachment to a net is illegal.

U_SET Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: U_SET
- Attribute Values: *name*

  name is the identifier of the set

VHDL Syntax

Place this constraint immediately before the module declaration or instantiation:

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*;

name is the identifier of the set

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Specify the Verilog constraint as follows:

(* U_SET = *name* ***)**

name is the identifier of the set

For more information on basic Verilog syntax, see Verilog .

UCF and NCF Syntax

The basic UCF syntax is:

INST "*instance_name***" U_SET=** *name***;**

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

BEGIN MODEL *entity_name*

INST "*instance_name***" U_SET=***uset_name***;**

END;

USE_RLOC (Use Relative Location)

USE_RLOC is an advanced mapping and placement constraint. It turns RLOC on or off for a specific element or section of a set.

USE_RLOC Architecture Support

This constraint applies to FPGA devices only.

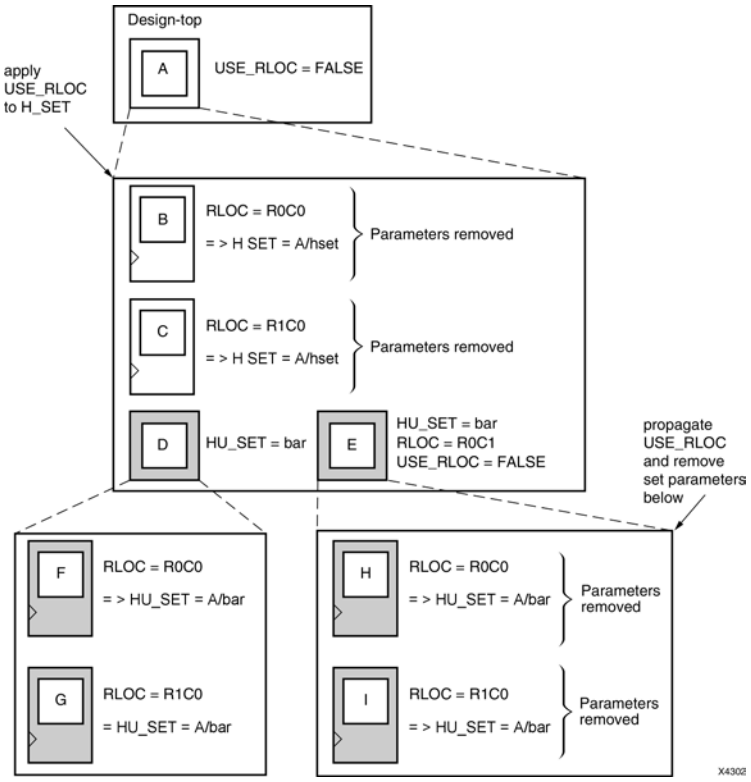
USE_RLOC Applicable Elements

Instances or macros that are members of sets

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.

Using the USE_RLOC Constraint to Control RLOC Application on H_SET and HU_SET Sets

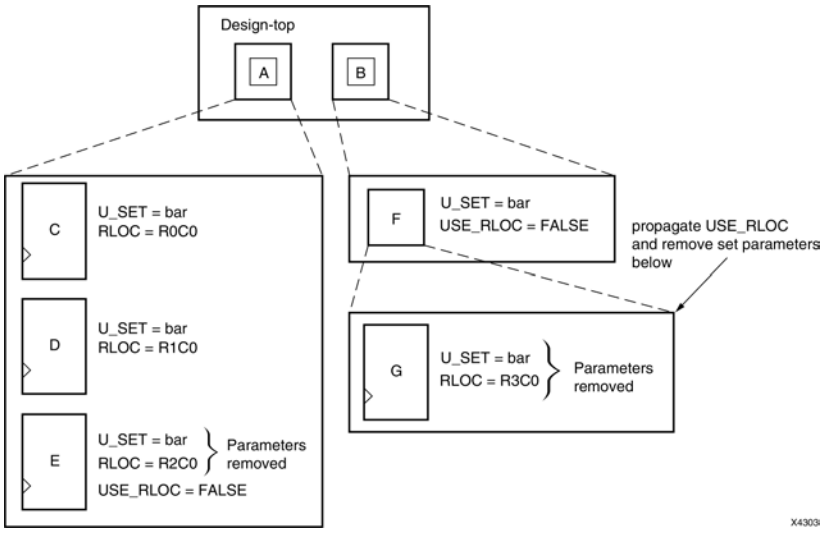


Applying the USE_RLOC constraint on U_SET sets is a special case because of the lack of hierarchy in the U_SET set. Because the USE_RLOC constraint propagates strictly in a hierarchical manner, the members of a U_SET set that are in different parts of the design hierarchy must be tagged separately with 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.

Using the USE_RLOC Constraint to Control RLOC Application on U_SET Sets



This illustration shows 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.

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.

USE_RLOC Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a member of a set
- Attribute Name: USE_RLOC
- Attribute Values: TRUE, FALSE

VHDL Syntax

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}";`

- **`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.

Verilog Syntax

Place this constraint immediately before the module declaration or instantiation:

Specify the Verilog constraint as follows:

(* USE_RLOC = "{TRUE|FALSE}" *)

- **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.

UCF and NCF Syntax

The basic UCF syntax is:

INST "*instance_name***" USE_RLOC={TRUE|FALSE};**

- **TRUE** turns on the RLOC constraint for a specific element
- **FALSE** turns it off

The default is **TRUE**.

XCF Syntax

MODEL "*entity_name***" use_rloc={true|false};**

VCCAUX (VCCAUX)

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 Architecture Support

This 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 Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

Placed on output or bidirectional port:

CONFIG VCCAUX="*value***";**

Where value is 2.5 or 3.3

Example:

CONFIG VCCAUX=3.3;

VOLTAGE (Voltage)

VOLTAGE is a 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 Architecture Support

The following FPGAs are supported for VOLTAGE:

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

VOLTAGE Applicable Elements

Global

VOLTAGE Propagation Rules

It is illegal to attach VOLTAGE to a net, signal, or design element.

VOLTAGE Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

UCF and NCF Syntax

VOLTAGE=*value* [**V**]**;**

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

From the Project Navigator Processes window, access the Constraints Editor by double-clicking Create Timing Constraints under User Constraints. Under Timing Constraints in the Constraint Type list box, double-click Operating Conditions to access a dialog box.

PCF Syntax

Same as UCF

VREF (VREF)

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 Architecture Support

This constraint applies only to CoolRunner-II devices with 128 macrocells and larger.

VREF Applicable Elements

Global

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

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

VREF=*value_list* (on CONFIG symbol)

Valid values are:

- P*nn*

  where

  nn is a numeric pin number
- *rc*

  where

  – *r*=alphabetic row

  – *c*=numeric column

UCF and NCF Syntax

CONFIG VREF=*value_list***;**

Valid values are:

- P*nn*

  where

  nn is a numeric pin number
- *rc*

  where

  r=alphabetic row

c=numeric column

```
CONFIG VREF=P12,P13;
```

WIREAND (Wire And)

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 Architecture Support

This constraint applies only to XC9500 devices.

WIREAND Applicable Elements

Any net

WIREAND Propagation Rules

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

WIREAND Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a net
- Attribute Name: WIREAND
- Attribute Values: TRUE, FALSE

VHDL Syntax

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.

Verilog Syntax

Place this constraint immediately before an instantiation:

Specify the Verilog constraint as follows:

```
(* WIREAND = "{YESE|NO|TRUE|FALSE}" *)
```

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

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)

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 slices. 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 slice. Using XBLKNM, you can partition a complete slice without constraining the slice to a physical location on the device.

Hierarchical paths are not prefixed to XBLKNM attributes, so XBLKNM attributes for different slices 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.

Note This constraint can be used with block RAMs also

XBLKNM Architecture Support

This 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 details. Also for more information, see the device data sheet.

XBLKNM Propagation Rules

Applies to the design element to which it is attached

XBLKNM Syntax

The following examples show how to use this constraint with particular tools or methods. If a tool or method is not listed, you cannot use this constraint with it.

Schematic Syntax

- Attach to a valid instance
- Attribute Name: XBLKNM
- Attribute Values: *block_name*

  block_name is a valid block name for that type of symbol

VHDL Syntax

Place this constraint immediately before the module declaration or instantiation:

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*`;`

block_name is a valid block name for that type of symbol

For more information on basic VHDL syntax, see VHDL.

Verilog Syntax

Specify the Verilog constraint as follows:

(* XBLKNM = "*block_name***" *)**

block_name is a valid block name for that type of symbol

For more information on basic Verilog syntax, see Verilog.

UCF and NCF Syntax

The basic UCF syntax is:

INST "*instance_name***" XBLKNM=***block_name***;**

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

BEGIN MODEL "*entity_name***"**

INST "*instance_name***" xblknm=***xblknm_name***;**

END;