IEEE STANDARDS ASSOCIATION

IEEE Standard for Design and Verification of Low-Power Integrated Circuits

IEEE Computer Society

Sponsored by the
Design Automation Committee
and the
IEEE Standards Association Corporate Advisory Group

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

29 May 2013

IEEE Std 1801™-2013
(Revision of IEEE Std 1801-2009)

IEEE Std 1801™-2013
(Revision of
IEEE Std 1801-2009)

IEEE Standard for Design and Verification of Low-Power Integrated Circuits

Sponsor

Design Automation Committee
of the
IEEE Computer Society
and the
IEEE Standards Association Corporate Advisory Group

Approved 6 March 2013

IEEE-SA Standards Board

Grateful acknowledgment is made to the following for permission to use source material:

Accellera Systems Initiative

Unified Power Format (UPF) Standard, Version 1.0

Cadence Design Systems, Inc.

Library Cell Modeling Guide Using CPF

Hierarchical Power Intent Modeling Guide Using CPF

Silicon Integration Initiative, Inc.

Si2 Common Power Format Specification, Version 2.0

Abstract: A method is provided for specifying power intent for an electronic design, for use in verification of the structure and behavior of the design in the context of a given power management architecture, and for driving implementation of that power management architecture. The method supports incremental refinement of power intent specifications required for IP-based design flows.
Keywords: corruption semantics, IEEE 1801™, interface specification, IP reuse, isolation, level-shifting, power-aware design, power domains, power intent, power modes, power states, progressive design refinement, retention, retention strategies

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Print: ISBN 978-0-7381-8282-7 STDPD98167
PDF: ISBN 978-0-7381-8281-0 STDGT98167

Notice and Disclaimer of Liability Concerning the Use of IEEE Documents: IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. IEEE develops its standards through a consensus development process, approved by the American National Standards Institute, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of the Institute and serve without compensation. While IEEE administers the process and establishes rules to promote fairness in the consensus development process, IEEE does not independently evaluate, test, or verify the accuracy of any of the information or the soundness of any judgments contained in its standards.

Use of an IEEE Standard is wholly voluntary. IEEE disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon any IEEE Standard document.

IEEE does not warrant or represent the accuracy or content of the material contained in its standards, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or fitness for a specific purpose, or that the use of the material contained in its standards is free from patent infringement. IEEE Standards documents are supplied "**AS IS**."

The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE standard is subjected to review at least every ten years. When a document is more than ten years old and has not undergone a revision process, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE standard.

In publishing and making its standards available, IEEE is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is IEEE undertaking to perform any duty owed by any other person or entity to another. Any person utilizing any IEEE Standards document, should rely upon his or her own independent judgment in the exercise of reasonable care in any given circumstances or, as appropriate, seek the advice of a competent professional in determining the appropriateness of a given IEEE standard.

Translations: The IEEE consensus development process involves the review of documents in English only. In the event that an IEEE standard is translated, only the English version published by IEEE should be considered the approved IEEE standard.

Official Statements: A statement, written or oral, that is not processed in accordance with the IEEE-SA Standards Board Operations Manual shall not be considered the official position of IEEE or any of its committees and shall not be considered to be, nor be relied upon as, a formal position of IEEE. At lectures, symposia, seminars, or educational courses, an individual presenting information on IEEE standards shall make it clear that his or her views should be considered the personal views of that individual rather than the formal position of IEEE.

Comments on Standards: Comments for revision of IEEE Standards documents are welcome from any interested party, regardless of membership affiliation with IEEE. However, IEEE does not provide consulting information or advice pertaining to IEEE Standards documents. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Since IEEE standards represent a consensus of concerned interests, it is important to ensure that any responses to comments and questions also receive the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to comments or questions except in those cases where the matter has previously been addressed. Any person who would like to participate in evaluating comments or revisions to an IEEE standard is welcome to join the relevant IEEE working group at http://standards.ieee.org/develop/wg/.

Comments on standards should be submitted to the following address:

Secretary, IEEE-SA Standards Board

445 Hoes Lane

Piscataway, NJ 08854

USA

Photocopies: Authorization to photocopy portions of any individual standard for internal or personal use is granted by The Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; +1 978 750 8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Notice to users

Laws and regulations

Users of IEEE Standards documents should consult all applicable laws and regulations. Compliance with the provisions of any IEEE Standards document does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

This document is copyrighted by the IEEE. It is made available for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making this document available for use and adoption by public authorities and private users, the IEEE does not waive any rights in copyright to this document.

Updating of IEEE documents

Users of IEEE Standards documents should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE-SA Website at http://standards.ieee.org/index.html or contact the IEEE at the address listed previously. For more information about the IEEE Standards Association or the IEEE standards development process, visit IEEE-SA Website at http://standards.ieee.org/index.html.

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http:/standards.ieee.org/findstds/errata/index.html. Users are encouraged to check this URL for errata periodically.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken by the IEEE with respect to the existence or validity of any patent rights in connection therewith. If a patent holder or patent applicant has filed a statement of assurance via an Accepted Letter of Assurance, then the statement is listed on the IEEE-SA Website at http://standards.ieee.org/about/sasb/patcom/patents.html. Letters of Assurance may indicate whether the Submitter is willing or unwilling to grant licenses under patent rights without compensation or under reasonable rates, with reasonable terms and conditions that are demonstrably free of any unfair discrimination to applicants desiring to obtain such licenses.

Essential Patent Claims may exist for which a Letter of Assurance has not been received. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims, or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

The Unified Power Format Working Group is entity based. At the time this standard was completed, the Unified Power Format Working Group had the following membership:

John Biggs, *Chair*
Erich Marschner, *Vice Chair*
Jeffrey Lee, *Secretary*
Joe Daniels, *Technical Editor*

Dave Allen
Ido Bourstein
Shir-Shen Chang
David Cheng
Cary Chin
Sumit DasGupta
Sorin Dobre
Shaun Durnan
Colin Holehouse
Sushma Honnavara-Prasad
Fred Jen
Tim Jordan
Knut Just
James Kehoe
Rick Koster
Rolf Lagerquist
Lisa McIlwain
Don Mills
Barry Pangrle
Albert Rich
Judith Richardson
Jim Sproch
Amit Srivastava
Prasad Subbarao
Venki Venkatesh
Qi Wang
Jon Worthington
Louis Yu

The following members of the entity balloting committee voted on this standard. Balloters may have voted for approval, disapproval, or abstention.

Accellera Systems Initiative
Advanced Micro Devices (AMD)
ARM, Ltd.
Atrenta Inc.
Broadcom Corporation
Cadence Design Systems, Inc.
Cambridge Silicon Radio
Cortina Systems
Intel Corporation
Japan Electronics and Information Technology Industries Association (JEITA)
LSI Corporation
Marvell Technology Group Ltd.
MediaTek Inc.
Mentor Graphics Corporation
Qualcomm Incorporated
Silicon Integration Initiative, Inc.
STMicroelectronics
Synopsys, Inc.
Texas Instruments Incorporated
Xilinx

When the IEEE-SA Standards Board approved this standard on 6 March 2013, it had the following membership:

John Kulick, *Chair*
David J. Law, *Vice Chair*
Richard H. Hulett, *Past Chair*
Konstantinos Karachalios, *Secretary*

Masayuki Ariyoshi
Peter Balma
Farooq Bari
Ted Burse
Wael William Diab
Stephen Dukes
Jean-Philippe Faure
Alexander Gelman
Mark Halpin
Gary Hoffman
Paul Houzé
Jim Hughes
Michael Janezic
Joseph L. Koepfinger*
Oleg Logvinov
Ron Petersen
Gary Robinson
Jon Walter Rosdahl
Adrian Stephens
Peter Sutherland
Yatin Trivedi
Phil Winston
Yu Yuan

*Member Emeritus

Also included are the following nonvoting IEEE-SA Standards Board liaisons:

Richard DeBlasio, *DOE Representative*
Michael Janezic, *NIST Representative*

Julie Alessi
IEEE Standards Program Manager, Document Development

Krista Gluchoski
IEEE Standards Program Manager, Technical Program Development

Introduction

This introduction is not part of IEEE Std 1801-2013, IEEE Standard for Design and Verification of Low-Power Integrated Circuits.

The purpose of this standard is to provide portable low-power design specifications that can be used with a variety of commercial products throughout an electronic system design, analysis, verification, and implementation flow.

When the electronic design automation (EDA) industry began creating standards for use in specifying, simulating, and implementing functional specifications of digital electronic circuits in the 1980s, the primary design constraint was the transistor area necessary to implement the required functionality in the prevailing process technology at that time. Power considerations were simple and easily assumed for the design as power consumption was not a major consideration and most chips operated on a single voltage for all functionality. Therefore, hardware description languages (HDLs) such as VHDL (IEC 61691-1-1/ IEEE Std 1076™)[a] and SystemVerilog (IEEE Std 1800™) provided a rich set of capabilities necessary for capturing the functional specification of electronic systems, but no capabilities for capturing the power architecture (how each element of the system is to be powered).

As the process technology for manufacturing electronic circuits continued to advance, power (as a design constraint) continually increased in importance. Even above the 90 nm process node size, dynamic power consumption became an important design constraint as the functional size of designs increased power consumption at the same time battery-operated mobile systems, such as cell phones and laptop computers, became a significant driver of the electronics industry. Techniques for reducing dynamic power consumption—the amount of power consumed to transition a node from a `0` to `1` state or vice versa—became commonplace. Although these techniques affected the design methodology, the changes were relatively easy to accommodate within the existing HDL-based design flow, as these techniques were primarily focused on managing the clocking for the design (more clock domains operating at different frequencies and gating of clocks when logic in a clock domain is not needed for the active operational mode). Multi-voltage power-management methods were also developed. These methods did not directly impact the functionality of the design, requiring only level-shifters between different voltage domains. Multi-voltage power domains could be verified in existing design flows with additional, straight-forward extensions to the methodology.

With process technologies below 100 nm, static power consumption has become a prominent and, in many cases, dominant design constraint. Due to the physics of the smaller process nodes, power is leaked from transistors even when the circuitry is quiescent (no toggling of nodes from `0` to `1` or vice versa). New design techniques were developed to manage static power consumption. Power gating or power shut-off turns off power for a set of logic elements. Back-bias techniques are used to raise the voltage threshold at which a transistor can change its state. While back bias slows the performance of the transistor, it greatly reduces leakage. These techniques are often combined with multi-voltages and require additional functionality: power-management controllers, isolation cells that logically and/or electrically isolate a shutdown power domain from "powered-up" domains, level-shifters that translate signal voltages from one domain to another, and retention registers to facilitate fast transition from a power-off state to a power-on state for a domain.

The EDA industry responded with multiple vendors developing proprietary low-power specification capabilities for different tools in the design and implementation flow. Although this solved the problem locally for a given tool, it was not a global solution in that the same information was often required to be specified multiple times for different tools without portability of the power specification. At the Design

[a]Information on references can be found in Clause 2.

Automation Conference (DAC) in June 2006, several semiconductor/electronics companies challenged the EDA industry to define an open, portable power specification standard. The EDA industry standards incubation consortium, Accellera Systems Initiative, answered the call by creating a Technical SubCommittee (TSC) to develop a standard. The effort was named Unified Power Format (UPF) to recognize the need of unifying the capabilities of multiple proprietary formats into a single industry standard. Accellera approved *UPF 1.0* as an Accellera standard in February 2007. In May 2007, Accellera donated UPF to the IEEE for the purposes of creating an IEEE standard, and in March 2009, the first version of the IEEE Std 1801 was released. So this standard, although the second version of the IEEE Std 1801, represents the third version of what is more colloquially referred to as UPF.

Contents

IEEE Standard for Design and Verification of Low-Power Integrated Circuits

1. Overview

1.1 Scope

This standard establishes a format used to define the low-power design intent for electronic systems and electronic intellectual property (IP). The format provides the ability to specify the supply network, switches, isolation, retention, and other aspects relevant to power management of an electronic system. The standard defines the relationship between the low-power design specification and the logic design specification captured via other formats [e.g., standard hardware description languages (HDLs)].

1.2 Purpose

The standard provides portability of low-power design specifications that can be used with a variety of commercial products throughout an electronic system design, analysis, verification, and implementation flow.

1.3 Key characteristics of the Unified Power Format

The Unified Power Format (UPF) provides the ability for electronic systems to be designed with power as a key consideration early in the process. UPF accomplishes this by allowing the specification of what was traditionally physical implementation-based power information early in the design process—at the register transfer level (RTL) or earlier. Figure 1 shows UPF supporting the entire design flow. UPF provides a

consistent format to specify power design information that may not be easily specifiable in an HDL or when it is undesirable to directly specify the power semantics in an HDL, as doing so would tie the logic specification directly to a constrained power implementation. UPF specifies a set of HDL attributes and HDL packages to facilitate the expression of power intent in HDL when appropriate (see Table 4 and Annex B). UPF also defines consistent semantics across verification and implementation, i.e., what is implemented is the same as what has been verified.

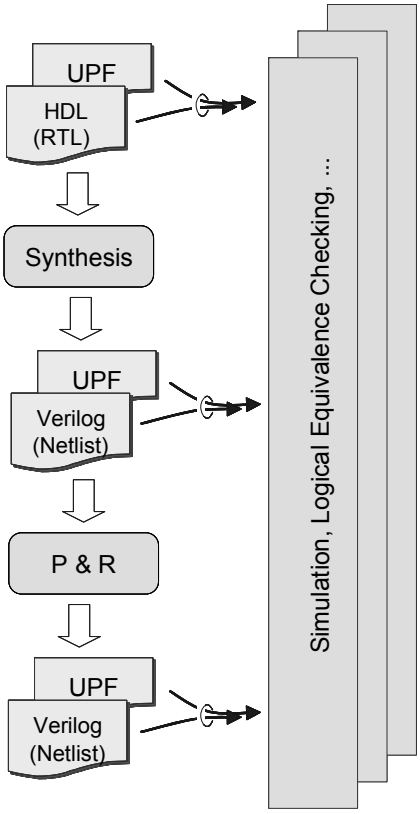

Figure 1—UPF tool flow

As indicated in Figure 1, UPF files are part of the design source and, when combined with the HDL, represent a complete design description: the HDL describing the logical intent and the UPF describing the power intent. Combined with the HDL, the UPF files are used to describe the intent of the designer. This collection of source files is the input to several tools, e.g., simulation tools, synthesis tools, and formal verification tools. UPF supports the successive refinement methodology (see 4.8) where power intent information will grow along the design flow to provide needed information for each design stage.

— Simulation tools can read the HDL/UPF design input files and perform RTL power-aware simulation. At this stage, the UPF may only contain abstract models such as power domains and supply sets without the need to create the power and ground network and implementation details.

— Synthesis tools can read the HDL/UPF design input files and produce a netlist. The tool or user may produce a new UPF fileset that, combined with the netlist, represents a further refined version of same design.

— In those cases where design object names change, a UPF file with the new names is needed. A UPF-aware logical equivalence checker can read the full design and UPF filesets and perform the checks to ensure power-aware equivalence.

— Place and route tools read both the netlist and the UPF files and produce a physical netlist, potentially including an output UPF file.

UPF is a concise power intent specification capability. Power intent can be easily specified over many elements in the design. A UPF specification can be included with the other deliverables of IP blocks and reused along with the other delivered IP. UPF supports various methodologies through carefully defined semantics, flexibility in specification, and, when needed, defined rational limitations that facilitate automation in verification and implementation (see Annex E).

A *UPF specification* defines how to create a supply network to supply power to each instance, how the individual supply nets behave with respect to one another, and how the logic functionality is extended to support dynamic power switching to these logic instances. By controlling the states and voltages of the supplies provided to the supply network, and by controlling the states of power switches that are part of the supply network, the power management logic of a system can cause each functional region to receive the power required to complete its computational tasks in a timely manner.

1.4 Use of color in this standard

This standard uses a minimal amount of color to enhance readability. The coloring is not essential and does not affect the accuracy of this standard when viewed in pure black and white. The places where color is used are the following:

- Cross references that are hyperlinked to other portions of this standard are shown in underlined-blue text (hyperlinking works when this standard is viewed interactively as a PDF file).
- Syntactic keywords and tokens in the formal language definitions are shown in **boldface-red text**.
- Command arguments that can be provided incrementally (*layered*) are shown in **boldface-green text**. See also 5.11.
- Syntactic keywords and tokens that have been explicitly identified as legacy or deprecated constructs (see 6.1) may be shown in **brown text**.

1.5 Contents of this standard

The organization of the remainder of this standard is as follows:

- Clause 2 provides references to other applicable standards that are presumed or required for this standard.
- Clause 3 defines terms and acronyms used throughout the different specifications contained in this standard.
- Clause 4 describes the basic concepts underlying UPF.
- Clause 5 describes the language basics for UPF and its commands.
- Clause 6 details the syntax and semantics for each UPF power intent command.
- Clause 7 details the syntax and semantics for each UPF power-management cell command.
- Clause 8 defines a reference model for UPF command processing.
- Clause 9 defines simulation semantics for various UPF commands.
- Annexes. Following Clause 9 are a series of annexes.

2. Normative references

The following referenced documents are indispensable for the application of this standard (i.e., they must be understood and used, so each referenced document is cited in the text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEC 61691-1-1/IEEE Std 1076™, Behavioural languages—Part 1: VHDL Language Reference Manual.[1, 2]

IEEE Std 1800™, IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.[3]

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary Online* [B1] should be consulted for terms not defined in this clause. [4, 5] Certain terms in this standard reflect their corresponding definitions in IEEE Std 1800 or IEC 61691-1-1/IEEE Std 1076, or they are listed in Annex A.[6]

3.1 Definitions

active component: A **component** that contains one or more input receivers and one or more output drivers whose values are functions of the inputs, but whose inputs and outputs are not directly connected; or any HDL construct(s) that synthesize(s) to an active component.

active control signal: A control signal that is currently presenting the value (level) or transition (edge) that enables or triggers an active component to operate in a particular manner.

active power state: A power state whose logic expression and, if present, supply expression evaluate to *True* at a given time.

activity: Any change in the value of a net, regardless of whether that change is propagated to an output.

ancestor: Any **instance** between the current **scope** in the **logic hierarchy** and its **root scope**. When the current **scope** is a top-level module, it does not have any ancestors. *See also:* **descendant**.

anonymous object: An object that is not named in the context of UPF. Implementations may assign a legal name, but such names are not visible in the UPF context.

balloon latch: A retention element style in which a register's value is saved to a dedicated latch at power-down and the latch value is restored to the register at power-up.

boundary instance: An **instance** that has no parent or whose **parent** is in a different **power domain**.

[1]IEC publications are available from the International Electrotechnical Commission (http://www.iec.ch/). IEC publications are also available in the United States from the American National Standards Institute (http://www.ansi.org/).

[2]IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).

[3]The IEEE standards or products referred to in this clause are trademarks of The Institute of Electrical and Electronics Engineers, Inc.

[4]*IEEE Standards Dictionary Online* subscription is available at:
http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

[5]The numbers in brackets correspond to those of the bibliography in Annex A.

[6]Information on references can be found in Clause 2.

component: A physical and logical construction that relates inputs to outputs.

composite domain: A power domain consisting of subordinate **power domains** called **subdomains**. All **subdomains** in a composite domain share the same **primary supply set**. Any operation performed on a composite domain has the same effect as performing the operation on each of its **subdomains**.

configuration UPF: The UPF specification for an intellectual property (IP) block that defines a particular configuration of the block for use in a given system. The configuration UPF typically includes the **constraint UPF** and extends it with configuration-specific details. Sometimes referred to as **golden UPF**.

connected: Attached together via a direct connection.

constraint UPF: The UPF specification for an intellectual property (IP) block that defines constraints that must be met by any configuration of the IP block used in a larger system. Sometimes referred to as **platinum UPF**.

corruption semantics: The rules defining the behavior of logic response to reduction or disconnection of power to that logic.

current scope: The design hierarchy location that serves as the immediate context for interpretation and execution of UPF commands. Also, the **instance** specified by the **set_scope** command.

NOTE—See 6.52.[7]

declared: Specified in the HDL explicitly or implicitly via a UPF command.

descendant: Any **instance** between the **current scope** in the **logic hierarchy** and its **leaf-level instance**s. When the **current scope** is a **leaf-level instance**, it does not have any descendants. *See also:* **ancestor**.

descendant subtree: A portion of a **logic hierarchy**, rooted at one **instance** in the hierarchy, and containing that **instance** and all of its **descendant**s.

design hierarchy: A hierarchical structure of nested definitions described in an HDL.

direct connection: A physical wire; or any HDL construct(s) that synthesize(s) to a direct connection.

domain port: A **port** that is on the interface of a **power domain**.

driver: The source or drain of a transistor, if the drain or source is connected to a power rail; a complementary metal oxide semiconductor (CMOS) inverter that continually connects a node to power or ground; any component that sets the value of its output via a transistor or inverter; a constant assignment; any combinational logic including a buffer of any kind; any sequential logic; or any HDL construct(s) that synthesize(s) to such combinational or sequential logic.

driver supply: For a **driver** that is a transistor, the supply connected to its source or drain; for a **driver** that is an inverter, the pair of supplies connected to the source/drain of the transistor pair comprising the inverter; or for an output of an **active component**, the related **supply set** of that output.

electrically equivalent: For **supply port**s/**net**s, connected (whether the connections are evident or not in the design) without any intervening switches, and therefore guaranteed to have the same value at all times from the perspective of any load; for **supply set**s/set handles, consisting of a set of electrically equivalent **supply net**s for each required function.

[7]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

equivalent: A pair of **supply net**s or a pair of **supply set**s that are considered to be interchangeable for certain purposes. *See also:* **electrically equivalent; functionally equivalent**.

erroneous: A usage that is likely to lead to an error in the design, but that tools may not be able to detect and report.

extent (of a domain): The set of instances that comprise a **power domain**.

feedthrough: A direct connection between two **port**s on the **interface of a power domain**, where the connection involves two **port**s on the upper boundary, or two **port**s on the lower boundary, or one of each; also, a direction connection between two **port**s of the same **leaf-level instance**.

feedthrough port: A **port** on the **interface of a power domain** that is part of a **feedthrough** through that domain, or a **port** on the interface of a **leaf-level instance** that is part of a **feedthrough** through that **instance**.

functionally equivalent: Functioning identically from the perspective of any load, either as a result of being **electrically equivalent**, or due to independent but parallel circuitry.

generate block: In the HDL code, this represents a level of design hierarchy, although a generate block is not itself an **instance**. After synthesis, generate blocks do not exist as an independent level of hierarchy. It is illegal to create any UPF objects in a **scope** that corresponds to a generate block.

golden source: The design together with the constraint UPF and the configuration UPF.

golden UPF: *See:* **configuration UPF**.

hard macro: A block that has been completely implemented and can be used as it is in other blocks. This may be modeled by an HDL module for verification or as a library cell for implementation.

hierarchical name: A series of names separated by the **hierarchical separator character**, the final name of which may be any legal HDL name or UPF name, and each preceding name is the name of an **instance** or **generate block** in which the following name is declared. *See also:* **hierarchical separator character**.

hierarchical separator character: A special character used in composing hierarchical names. The hierarchical separator character is a slash (/).

HighConn: The side of a **port** connection that is higher in the **design hierarchy**; the actual signal associated with a formal **port** definition.

implementation UPF: The UPF specification of how power distribution and control is to be implemented for a system. The implementation UPF typically includes the configuration UPF for each of the intellectual property (IP) blocks instantiated in the system. Sometimes referred to as **silicon UPF**.

inactive: A normally active component in a state in which it does not respond to **activity** on its inputs. Also, a control signal that is not currently presenting the value (level) or transition (edge) that enables or triggers an **active component** to operate in a particular manner.

instance: A particular occurrence of a SystemVerilog module (see IEEE Std 1800), VHDL entity (see IEC 61691-1-1/IEEE Std 1076), or library cell at a specific location within the design hierarchy.

interface of a power domain: The union of the **upper boundary** and the **lower boundary** of the **power domain**.

isolation: A technique used to provide defined behavior of a logic signal when its driving logic is not active.

isolation cell: An **instance** that passes logic values during normal mode operation and clamps its output to some specified logic value when a control signal is asserted.

leaf-level cell: An **instance** that has no **descendant**s, or an **instance** that has the attribute **UPF_is_leaf_cell** associated with it.

NOTE—See Table 4.

leaf-level instance: *See:* **leaf-level cell**.

level-shifter: An **instance** that translates signal values from an input voltage swing to a different output voltage swing.

live slave: A retention element style in which the slave latch of a master-slave flip-flop (MSFF) is always on and therefore maintains the value of the MSFF during power-down.

logic hierarchy: An abstract view of a **design hierarchy** in which only those definitions representing **instance**s are included.

LowConn: The side of a **port** connection that is lower in the **design hierarchy**; the formal **port** definition.

lower boundary (of a power domain): The HighConn side of each **port** of each **boundary instance** in the **extent** of another **power domain** whose **parent** is in the **extent** of this domain, together with the HighConn side of each **port** of any macro cell instance in this **power domain**, for which the related supply set is neither identical to nor equivalent to the **primary supply set** of this domain.

map: Identifies a specific **model** corresponding to an abstract behavior. An **instance** of the **model** can then be used to implement the specific behavior.

model: A SystemVerilog module, VHDL entity/architecture, or Liberty cell.

named power state: A **power state** defined using **add_power_state**, **add_port_state**, or **add_pst_state** for a **supply set** or **power domain**, or the **DEFAULT_NORMAL** and **DEFAULT_CORRUPT** power states predefined for supply sets.

net: The individual **net segment**s that make up a collection of interconnections between a collection of **port**s. A **net** may be named or anonymous.

net segment: A direct connection within a single **instance**.

parent: The immediate **ancestor** of a given **instance** within the **logic hierarchy**.

passive component: A direct connection; a **component** that has neither a **receiver** nor a **driver**, whose output is connected to its input, and therefore its output is always the same as its input, e.g., a pass transistor; or any HDL construct(s) that synthesize(s) to a **feedthrough** component.

pg_type: An attribute of a port that indicates its use in providing power to a cell.

platinum UPF: *See:* **constraint UPF**.

port: A **connection** on the interface of a SystemVerilog module or VHDL entity. Also, a **port** on the **interface of a power domain**.

power domain: A collection of **instance**s that are treated as a group for power-management purposes. The **instance**s of a power domain typically, but do not always, share a **primary supply set**. A power domain may also have additional supplies, including **retention** and **isolation** supplies.

power rail: The physical implementation of a power supply net.

power state: The state of a **supply net**, **supply port**, **supply set**, or **power domain**.

NOTE—See Clause 4.

power state table (PST): A table that captures the legal combinations of **power state**s for a set of **supply port**s and/or **supply net**s.

primary supply set: The **supply net** connections inferred for all **instance**s in the **power domain**, unless overridden.

receiver: The gate of a transistor; the input to an inverter; any **component** whose behavior is determined by an input signal; any combinational logic including a buffer of any kind; any sequential logic; or any HDL construct(s) that synthesize(s) to such combinational or sequential logic.

receiver supply: For a **receiver** that is the gate of a transistor, the supply connected to that transistor's source or drain; for a **receiver** that is the input to an inverter, the pair of supplies connected to the source/drain of the transistor pair comprising the inverter; or for a **receiver** that is part of an **active component**, the primary supply of the **power domain** to which that **receiver** belongs or, in some cases, the secondary supply of the **component** if it has a secondary supply.

regulator: An **instance** that takes a set of input **supply net**s and provides the source for a set of output **supply net**s. The output voltage is a function of the input voltages and the logical state of any control signals.

retention: Enhanced functionality associated with selected **sequential element**s or a memory such that memory values can be preserved during the power-down state of the primary supplies.

retention register: A register that extends the functionality of a **sequential element** with the ability to retain its memory value during the power-down state.

rooted name: The **hierarchical name,** relative to the **current scope**, of an object in the **logic hierarchy** or a UPF object defined for a **scope** in the hierarchy.

root scope: The topmost scope in the **logic hierarchy**, which contains an implicit instance of each top-level module.

root supply driver: The origin of a supply, e.g., an on-system voltage regulator, bias generator modeled in HDL, or an off-chip supply source. *See also:* **supply source**.

root supply source: An input or inout **supply port** that is not connected to an "upstream" **supply net**; an input or inout **supply port** that is not connected to a root supply source defined in an ancestor scope; an output or inout **supply port** that is not connected to a **supply source** defined in a child scope; a **supply set** or supply set handle function that is neither associated with a **supply port** or **supply net** (via **associate_supply_set**) nor connected to another root supply source (via **connect_supply_net**).

NOTE—See 6.7 and 6.11.

scope: An **instance** in the **logic hierarchy**.

silicon UPF: *See:* **implementation UPF**.

simple name: An identifier that denotes an object declared in a given **scope** and is not a **hierarchical name**.

simstate: The level of operational capability supported by a given **power state** of a **supply set**.

sink: A **receiver**; the HighConn of an input port or inout port of an **instance**; or the LowConn of an output port or inout port of an **instance**.

source: A **driver**; the LowConn of an input port or inout port of an **instance**; or the HighConn of an output port or inout port of an **instance**.

state element: A sequential element such as a flip-flop, latch, or memory element. Also, a conditionally stored value in register transfer level (RTL) code from which a sequential element would be inferred.

strategy: A rule that specifies where and how to apply isolation, level-shifting, state retention, and buffering in the implementation of power intent.

subdomain: A member of the set of domains comprising a composite power domain.

supply function: An abstraction of a **supply net** in a **supply set**, the name of which identifies the purpose of the corresponding **net** in the **supply set**.

supply net: An HDL representation of a power rail.

supply port: A connection point for **supply net**s.

supply set: A collection of **supply function**s that in aggregate provide a complete power source.

supply source: A **supply port** that propagates but does not originate a supply value.

switch: An **instance** that conditionally connects one or more input **supply net**s to a single output **supply net** according to the logical state of one or more control inputs.

top-level instance: An implicit **instance** corresponding to a top-level module.

upper boundary (of a power domain): The LowConn side of each **port** of each **boundary instance** in the **extent** of this **power domain**.

3.2 Acronyms and abbreviations

CMOS complementary metal oxide semiconductor

DFT Design for Test

EDA electronic design automation

HDL hardware description language

IP intellectual property

MSFF master-slave flip-flop

NMOS N-channel metal oxide semiconductor

PG power/ground

PMOS P-channel metal oxide semiconductor

PST power state table

ROM Read-only Memory

RTL register transfer level

SAIF Switching Activity Interchange Format

SoC System On Chip

Tcl Tool Command Language

UPF Unified Power Format

VCT value conversion table

VHDL VHSIC hardware description language

VHSIC Very High Speed Integrated Circuit

4. UPF concepts

This clause provides an overview of concepts involved in defining power intent using UPF. These concepts include those related to the representation of the design structure and functionality in one or more hardware description languages (HDLs), as well as those related to power-management structures and functionality defined for and/or added to the design to model intended power-management capabilities.

The structure and functionality of a design is specified using HDLs such as Verilog [B2], SystemVerilog, or VHDL. Each HDL may have specific terminology and concepts that are unique to that language, but all HDLs share some common concepts and capabilities. A typical design may be expressed in one or more HDLs.

UPF is defined in terms of a generalized abstraction of an HDL-based design hierarchy. This abstraction enables the UPF definition to apply to a design expressed in any of the three HDLs previously mentioned, or in any combination thereof, while at the same time minimizing the complexity of the UPF definition. This clause presents the abstract model and maps it to specific HDL concepts.

UPF is intended to apply to a design as its representation changes from an abstract functional model to a concrete physical model, during which process the power intent expressed in UPF becomes realized as part of the implementation. Because of this, the abstract logic hierarchy that is the basis of the UPF definition shall be understood in terms of both functional specification and physical implementation.

4.1 Design structure

4.1.1 Transistors

At the lowest level, UPF focuses on controlling power (or more precisely, voltage and current) delivered to transistors. These are usually assumed to be digital complementary metal oxide semiconductor (CMOS) transistors, but they could be analog devices as well or implemented in other technologies. The gate connection of a transistor is a receiver; the source of the signal provided to a gate (in CMOS, typically the output of a `P/N` transistor pair) is a driver.

4.1.2 Standard cells

Transistors are seldom modeled individually in an HDL description; typically, collections of transistors are represented by standard cells that have been developed as part of a particular technology library, which is usually expressed in the Liberty library format (see [B7]). Such cells typically have a primary supply (power and ground) and may also have a secondary supply for related behavior (e.g., state retention).

4.1.3 Hard macros

A library may also contain hard macros, which provide predefined physical implementations for much larger and more complex functions. A hard macro may have multiple supplies.

4.2 Design representation

4.2.1 Models

Library elements have corresponding behavioral models for use in simulation. These models may or may not include power and ground pins for their supplies. Standard cell models are usually written as Verilog modules and use constructs such as Verilog built-in primitives or user-defined primitives (UDPs) to express the relatively simple behavior of a standard cell. They may also be written as VHDL design entities (entity/

architecture pairs) using package `VITAL`, which provides Verilog-like primitive modeling capabilities. Hard macro models may be written in either language, using more complex behavioral constructs such as Verilog initial blocks and always blocks or VHDL processes and concurrent statements.

4.2.2 Netlist

A netlist is a collection of unique instances of standard cells and hard macros, interconnected by nets (Verilog) or signals (VHDL). Such instances are considered to be leaf-level instances, because their models are not constructed from an interconnection of subordinate instances, but instead are built using behavioral or functional HDL statements. A netlist may also include hierarchical instances, i.e., instances of a model that is itself defined as a netlist.

A power/ground (PG) netlist is a netlist containing cell and/or hard macro instances that include power and ground pins and a representation of the power and ground supply routing for those instances. A non-PG netlist is one that does not include any representation of the power supply network.

4.2.3 Behavioral models

Behavioral models that are written using the RTL synthesis subset of Verilog or VHDL are synthesizable models, or *soft macros*, which can be read by an RTL synthesis tool and mapped to a functionally equivalent netlist. Synthesis involves identifying or inferring the state elements needed to implement the specified behavior and implementing the combinational logic interconnecting those elements and the model's ports.

For many synthesizable HDL constructs, synthesis creates combinational or sequential logic elements that are ultimately defined in terms of transistors, which in turn define drivers and receivers. In particular, any synthesizable statement that involves conditional computation or conditional updating of an output will most likely create logic. In contrast, unconditional assignment statements and port associations typically result in interconnect, not logic; for such HDL constructs, no drivers or receivers are created. In particular, ports do not create drivers; it is the logic driving a port that creates a driver for the port and for the net associated with the port.

4.2.4 HDL scopes

An HDL model defines one or more scopes. A scope is a region of HDL text within which names can be defined. Such names are typically visible (i.e., can be referenced) within the scope in which they are defined and, in certain cases, in other scopes (e.g., nested scopes). A Verilog model usually defines a single scope for the whole model. A VHDL model often defines multiple scopes; one for the whole model, plus other nested scopes for process statements and block statements. `generate` statements in either HDL are also considered to be nested scopes within the model's top-level scope.

4.2.5 Design hierarchy

A design hierarchy is constructed by defining one model in terms of interconnected instances of other models. Each instance represents a subtree of the hierarchy; the boundary between this subtree and its parent instance is defined by the interface of the model that has been instantiated to create the subtree. The interface consists of the model's ports, together with the nets associated with those ports for the instance that created this subtree. In Verilog, a port is defined as having two sides: a *HighConn* and a *LowConn*. The LowConn represents the port declaration in the model; the HighConn represents an instance of that port associated with an instance of the model, and therefore indirectly the net attached to that port instance. In VHDL, a somewhat different distinction is made between a "formal" port of a model and the "actual" signal associated with that port for a given instance of the model. In the context of UPF, regardless of what HDL is involved, the term LowConn means the (formal) port declaration in the model definition, and the term HighConn means the port of an instance of a model and by extension the net or signal connected to that port.

An HDL model that is not instantiated in any other instance is a top model, or simply `top`. A given design hierarchy usually contains a single top, but it may contain multiple tops in certain cases (e.g., if the design and the testbench in a simulation are modeled separately—neither instantiates the other). Each top is considered to be implicitly instantiated within the *root scope*. In Verilog, the root scope is `$root`; in VHDL, the root scope is the *root declarative region*. The instance name of such an implicit instance is the same as the model name.

4.2.6 Logic hierarchy

UPF assumes a somewhat more abstract model of the design hierarchy. This abstract model is called the *logic hierarchy*. As usual, the topmost scope is still the root scope and modules that are not instantiated elsewhere are the top modules (and instances) of the hierarchy. However, in the logic hierarchy, each scope corresponds to a whole instance; internal scopes presented in the design hierarchy are not modeled. In particular, HDL `generate` statements, which are considered to be internal scopes in the respective language definitions, are assumed to be collapsed into the parent module scope in the logic hierarchy.

UPF generally allows references to the names of objects defined anywhere in the subtree descending from a given instance, when the *current scope* is set to that instance. Such references are called *rooted names*, meaning they are hierarchical names relative to the current scope. If the design hierarchy contains `generate` statements that have been collapsed in the logic hierarchy, then the hierarchical name of an object in the logic hierarchy may include simple names that encode the collapsed scope names.

UPF also uses the logic hierarchy as a framework for locating the power-management objects used to represent power-management concepts, e.g., power domains and power state tables (PSTs). Each such object is effectively declared in a specific scope of the logic hierarchy, and the name of the scope can be used as the prefix of the name of the object.

The logic hierarchy can be viewed as a purely conceptual structure that is independent of the eventual physical implementation. Alternatively, the logic hierarchy can be viewed as an indication of the floor plan to be used in the physical implementation. Either view can be used, but it is best to adopt one view or the other for a given design, because the choice can affect how the power intent is expressed in UPF.

4.2.7 Hierarchy navigation

In UPF, commands are executed in the context of a scope within the logic hierarchy. The **set_scope** command (see 6.52) is used to navigate within the hierarchy and to set the current scope within which commands are executed.

Consistent with SystemVerilog `$root`, the root of the logic hierarchy is the scope in which the top modules are implicitly instantiated. Other locations within the logic hierarchy are referred to as the *design top instance*, which has a corresponding *design top module*, and the current scope.

The design top instance and design top module are typically paired: the design top instance (represented by a hierarchical name relative to the root scope) is an instance in the hierarchy representing a design for which power intent has been defined, and the design top module is the module for which the UPF file expressing this power intent has been written. The association between the UPF file and the design top module is specified in the UPF file using **set_design_top** (see 6.38); this UPF file is then typically applied to each instance of that module in a larger system.

The current scope is an instance that is, or is a descendant of, the design top instance (represented by a relative path name from the design top instance).

The **set_scope** command (see 6.52) changes the current scope locally within the subtree depending on the current design top instance/module. Since the design top instance is typically an instance of the design top

module, they both have the same hierarchical substructure; therefore, **set_scope** can be written relative to the module, but still work correctly when applied to an instance. The **set_scope** command is only allowed to change scope within this subtree. It cannot scope above the current design top instance.

The design top instance and design top module are initially set by the tool, possibly with direction from the user. They can be changed by invoking **load_upf** with the **–scope** argument (see 6.28). The current scope is reset whenever the design top instance changes. When **load_upf** completes, all three variables revert back to their previous settings.

4.2.8 Ports and nets

Ports define connection points between adjacent levels of hierarchy. In HDL, ports are defined as part of the interface of a module and therefore exist for each instance of the module. Nets define interconnections between a collection of ports. In HDL, nets are defined within a module and therefore exist within each instance of the module.

A port has two sides. The top side is the HighConn side, which is visible to the parent of the instance whose interface contains the port. The bottom side is the LowConn side, which is visible internal to the instance whose interface contains the port.

When a net in the current scope is connected to a port on a child instance, the connection is made to the HighConn side of the port. When a net in the current scope is connected to a port defined on the interface of the instance that is the current scope, the connection is made to the LowConn side of the port.

A port can be referenced wherever a net is required. Such a reference refers to the LowConn side of the port. A port can be thought of as being implicitly connected to an implicit net created with the same name and in the same scope as the LowConn side of the port.

4.2.9 Connecting nets to ports

In an HDL description, ports are typically required to pass nets from one level of hierarchy to another. In UPF, a net in the current scope can be connected to the LowConn of any port declared in the same scope or to the HighConn of any port within its descendant subtree. If the port is not declared in the same scope as the net, additional ports, nets, and port/net associations may be created to establish the connection from the net to the port. Such implicitly created ports and nets shall have the same simple name as the net being connected unless that name conflicts with the name of an existing port or net; in which case, to avoid a name conflict, the tool shall create a name that is unique for that scope.

NOTE—Nets are propagated as necessary through the descendant subtree and may be renamed to avoid name collision; therefore, the same simple name in different scopes may refer to nets that are independent and unconnected.

Implicitly created ports and nets should not be referenced directly by UPF commands, since the names of such ports and nets are not guaranteed to be the same as the original net name. These implicitly created ports and nets are merely a method of implementing a UPF connection in terms of valid HDL connections, when the UPF-specified power intent is represented in HDL form.

4.3 Power architecture

A UPF power intent specification defines the power architecture to be used in managing power distribution within a given design. The power architecture defines how the design is to be partitioned into regions that have independent power supplies, and how the interfaces between and interaction among those regions will be managed and mediated.

4.3.1 Power domains

A power domain is a collection of instances that are typically powered in the same way. In the physical implementation, the instances of a power domain are typically placed together and powered by the same power rails. In the logic hierarchy, the instances of a power domain are typically part of the same subtree of the hierarchy, or of sibling subtrees with a common ancestor, and powered by the same supply nets.

A power domain is defined within a scope (or instance) in the logic hierarchy. The definition of the power domain identifies the uppermost instances of the domain: those that define the upper boundary of the domain. For any given instance included in the power domain, a child instance of the given instance is transitively included in the power domain, unless that child instance is explicitly excluded from this power domain or is explicitly included in the definition of another power domain.

More formally, a boundary instance of a given power domain is any instance that has no parent (it is an implicit instance of a top-level module) or whose parent is in the extent of a different power domain. It is possible for one boundary instance of a power domain to be an ancestor of another boundary instance of the same power domain. This occurs when one instance is in the extent of a given power domain and both an ancestor and a descendant of that instance are in the extent of a second power domain. In this case, both the ancestor and the descendant may be boundary instances of the second domain. A domain with such a structure is referred to as a *donut* power domain.

The upper boundary of a power domain consists of the LowConn side of each port on each boundary instance in the domain. The lower boundary of a domain consists of the HighConn side of each port on each child instance that is in some other power domain or is a port of a macro cell instance that is powered differently from the rest of the domain. Both boundaries include any logic ports added to the design for power management. The interface of a power domain consists of the upper boundary and the lower boundary.

The instance in the logic hierarchy in which a power domain is defined is called the *scope* of the power domain. The set of instances that belong to a power domain are said to be the extent of that power domain. This distinction is important: while a given instance can be the scope of multiple power domains, it can be in the extent of one and only one power domain. As a consequence of these definitions, all instances within the extent of a domain are necessarily within the scope of the domain or its descendants.

A power domain can be either contiguous or non-contiguous. In the physical implementation, a contiguous power domain is one in which all instances are placed together; a non-contiguous power domain is one in which instances in the domain are placed in two or more disjoint locations. A power domain is contiguous within the logic hierarchy if it contains a single boundary instance; it is non-contiguous within the logic hierarchy if it includes multiple boundary instances.

For a non-contiguous power domain, a connection from an instance in the extent of the power domain to some other instance in the extent of the domain may need to be routed through another power domain.

Power domains that share a primary supply set may be composed together to form a larger power domain such that operations performed on this larger power domain apply transitively to each subdomain. In this way, unnecessary power domains may be aggregated together and handled as one for simplicity.

After UPF-specified power intent has been completely applied, it is an error if any instance is not included in a power domain.

4.3.2 Drivers, receivers, sources, and sinks

A logic signal in the design originates at an active component (the driver) and terminates at another active component (the receiver). Along the way it may pass through ports and nets. The driver and any port it

passes through on the way to a receiver is considered a source; the receiver and any port it passes through on the way from the driver is considered a sink. For example, a buffer defines both a source and a sink: the buffer's output port is a source; the buffer's input port is a sink.

A signal traversing a power domain may or may not be driven within the power domain. A port is neither a driver nor a receiver; it merely propagates a signal across a hierarchy boundary. If a port on the interface of a power domain is connected directly to another port on the interface of the same power domain, without going through an active component, the connection between those two ports has neither a driver nor a receiver in that domain. In this case, the connection is a feedthrough path through that domain.

HDL assignment statements may include delays, which may represent inertial delay (resulting from transistor switching) or transport delay (resulting from propagation along a wire). However, synthesis tools typically ignore such delays; therefore, the inclusion of such a delay, whether inertial or transport, does not by itself imply that an active component will be inferred from the assignment. For this reason, delays are not considered to create drivers or receivers.

A connection may be thought to "exist" in a given domain, if a user so chooses, but since a connection is by definition a passive component, it has no driver in the domain in which it exists and therefore is not affected or corrupted by the power state of the domain in which it exists.

4.3.3 Isolation and level-shifting

Two power domains interact if one contains logic that is the driver of a net and the other contains logic that is a receiver of the same net. When both power domains are powered up, the receiving logic should always see the driving logic's output as an unambiguous `1` or `0` value, except for a very short time when the value is in transition. The structure of CMOS logic typically ensures that minimal current flow will occur when the input value to a gate is a `1` or `0`. However, if the driving logic is powered down, the input to the receiving logic may float between `1` or `0`. This can cause significant current to flow through the receiving logic, which can damage the circuit. An undriven input can also cause functional problems if it floats to an unintended logic value.

To avoid this problem, isolation cells are inserted at the boundary of a power domain to ensure that receiving logic always sees an unambiguous `1` or `0` value. Isolation may be inserted for an input or for an output of the power domain. An isolation cell operates in two modes: normal mode, in which it acts like a buffer, and isolation mode, in which it clamps its output to a defined value. An isolation enable signal determines the operational mode of an isolation cell at any given time.

Two interacting power domains may also be operating with different voltage ranges. In this case, a logic `1` value might be represented in the driving domain using a voltage that would not be seen as an unambiguous 1 in the receiving domain. Level-shifters are inserted at a domain boundary to translate from a lower to a higher voltage range, and sometimes from a higher to a lower voltage range as well. The translation ensures the logic value sent by the driving logic in one domain is correctly received by the receiving logic in the other domain.

Isolation and level-shifting are often implemented in combination, so one standard cell implements both functions. UPF includes support for such "combo" cells.

Isolation and level-shifter strategies specify that isolation and level-shifter cells are to be inserted in specified locations. However, there are some cases where implementation tools may choose not to insert such cells, or to optimize redundant insertion of such cells. For example, isolation/level-shifters on floating ports that appear to have no drivers or have constant drivers may be removed or transformed, provided the resulting behavior is unchanged. To prevent implementation tools from applying such optimizations, isolation and level-shifting strategies can instead specify the respective cells are to be inserted regardless of optimization possibilities.

4.3.4 State retention

State retention is the ability to retain the value of a state element in a power domain while switching off the primary power to that element, and being able to use the retained value as the functional value of the state element upon power-up. State retention can enable a power domain to return to operational mode more quickly after a power-down/power-up sequence and it can be used to maintain state values that cannot be easily recomputed on power-up. State retention can be implemented using retention memories or retention registers. Retention registers are sequential elements (latches or flip-flops) that have state retention capability.

For a retention register, the following terms apply:

- *Register value* is the data held in the storage element of the register. In functional mode, this value gets updated on the rising/falling edge of clock or gets set or cleared by set/reset signals, respectively.
- *Retained value* is the data in the retention element of retention register. The retention element is powered by the retention supply.
- *Output value* is the value on the output of the register.

Depending on how the retained value is stored and retrieved, there are at least two flavors of retention registers, as follows:

a) *Balloon-style retention*: In a balloon-style retention register, the retained value is held in an additional latch, often called the *balloon latch*. In this case, the balloon element is not in the functional data-path of the register.

b) *Master/slave-alive retention*: In a master/slave-alive retention register, the retained value is held in the master or slave latch. In this case, the retention element is in the functional data-path of the register.

A balloon-style retention register typically has additional controls to transfer data from a storage element to the balloon latch, also called the *save step*, and transfer data from the balloon latch to the storage element, also called the *restore step*. The ports to control the save/restore pins of the balloon style retention register need to be available in the design to describe and implement this style of registers.

A master/slave-alive retention register typically does not have additional save/restore controls as the storage element is the same as the retention element. Additional control(s) on the register may park the register into a quiescent state and protect some of the internal circuitry during power-down state, and thus ensure the retention state is maintained. The restore in such registers typically happens upon power-up, again owing to the storage element being the same as the retention element. Thus, this style of registers may not specify save/restore signals, but may specify a retention condition that could take the register in and out of retention.

4.4 Power distribution

The electric current transported by a supply net originates at a root supply driver, which may be an on-chip voltage regulator, a bias generator modeled in HDL, or an off-chip supply source. A root supply driver's value may be conditionally propagated by a switch (modeled in HDL or created in UPF, see 6.18).

A root supply source (see 3.1) has an implicit root supply driver associated with it. Initially, the root supply driver drives the root supply source with the value `{OFF, unspecified}`. The package UPF functions `supply_on` and `supply_off` may be called to change the driving value of the root supply driver that drives a root supply source. It shall be an error if either of these functions is applied to an object that is not a root supply source. A root supply driver may also be created and manipulated via functions defined in package UPF (see Annex B).

A supply net may have one or more supply sources, depending upon its resolution type. During UPF processing, if the number of sources connected to a supply net do not conform to the requirements of its resolution type, an error shall be reported. At any given time during simulation, if the sources of a supply net do not conform to the requirements of its resolution type, the resolved value of the supply net at that time is set to `{UNDETERMINED, unspecified}`.

A power switch may have one or more input supply ports and one output supply port. Each input supply port may have one or more state definitions. At any given time during simulation, if the state definitions of a given input supply port are contradictory, or if multiple incompatible inputs are enabled at the same time, or if any input supply port is in an error state, the resolved value of the output supply port at that time is set to `{UNDETERMINED, unspecified}`.

The semantics defined in this standard, such as the supply net resolution functions, presume an idealized supply network with no voltage drop; the semantics for supply network resolution with modeled-voltage drop are outside the scope of this standard.

4.4.1 Supply network elements

Supply network objects (supply ports, supply nets, and switches) are created within the logic hierarchy to provide connection points for a root supply and to propagate the value of a root supply throughout a portion of the design. Supply network objects are created independent of power-domain definitions. This allows sharing of common components of the supply distribution network across multiple power domains.

4.4.1.1 Supply ports and nets

Supply ports provide a connection point for supply nets where they cross a hierarchy boundary. Supply nets can be used to create a connection between two supply ports or from a supply port to an instance within a power domain.

Supply ports and nets may be created in UPF or in the HDL design. If created in the HDL, the port or net shall be of the supply net type defined in the appropriate package UPF (see Annex B).

4.4.1.2 Supply switches

Supply switches conditionally propagate the value on an input supply port to an output supply port, depending upon the value of a control signal. A supply net may be connected to one or more power switches or supply ports, which may be connected to one or more root supply drivers.

4.4.1.3 Supply sets

A supply set represents a collection of supply nets that provide a complete power source for one or more instances. Each supply set defines six standard functions: **power**, **ground**, **pwell**, **nwell**, **deeppwell**, and **deepnwell**. Each function represents a potential supply net connection to a corresponding portion of a transistor. Each function of a given supply set can be associated with a particular supply net that implements the function.

A global supply set is one that is defined in a given scope and associates supply nets with its functions. One or more local supply sets, called *supply set handles*, can be defined for a power domain, a power switch, an isolation strategy (see 6.41), a level-shifting strategy (see 6.43), or a retention strategy (see 6.49). A supply set can be associated with a supply set handle as a whole; the functions of a supply set handle can be broken out and connected to ports of instances. This association creates a connection between the supply nets represented by corresponding functions of the supply set and supply set handle.

A supply set function is equivalent to a supply net and may be used anywhere a supply net is allowed. The supply set function represents the supply net that is or will be associated with that function of the supply set. The supply set function reference is a symbolic name for the supply net it represents.

A reference to a supply net by its symbolic name is an indirect reference.

NOTE—A supply net may be associated with a function of more than one supply set. The function that a given supply net performs in one supply set is unrelated to the function it may perform in any other supply set.

4.4.2 Supply network construction

Supply ports and nets are interconnected to create a supply network. Certain definitions and restrictions constrain how these interconnections are made.

4.4.2.1 Supply sources and loads

Supply ports define supply sources and supply loads, as follows:

- The LowConn of an input or inout port is a supply source. The HighConn of an output or inout port is a supply source (including a switch output).
- The LowConn of an output or inout port is a load. The HighConn of an input or inout port is a load (including a switch input).

A port that is neither a top-level port nor a leaf-level port is an internal (hierarchical) port.

4.4.2.2 Supply port/net connections

Connections are made from nets to ports

a) from a net to (the LowConn of) a port declared in the same scope; or
b) from a net to (the HighConn of) a port declared in a lower scope; or
c) from a net to a pin of a leaf cell.

The LowConn of a port may be used as an implicit net and connected to another port.

Only one net connection can be made to the LowConn of a port. Likewise, only one net connection can be made to the HighConn of a port. A source can be connected to a net that is in turn connected to multiple loads.

4.4.2.3 Supply net resolution

A supply net may be unresolved or resolved, as follows:

- An unresolved supply net shall have only one supply source connection.
- A resolved supply net may have multiple supply source connections. The resolution type may restrict how many supply sources can be on at the same time.

A supply net may have any number of load connections.

4.4.2.4 Supply net / supply set connections

Related supply nets can be grouped into a supply set, with each supply net in the group providing one or more functions of the supply set. The supply net corresponding to a given function of a supply set can be specified when the supply set is created or updated (see 6.22). One supply set may be associated with another supply set (see 6.7); this implicitly connects corresponding functions together and therefore it also

implicitly connects the supply nets associated with corresponding functions and any instance ports to which those functions are connected.

4.4.2.5 Supply set function connections

Supply functions of a supply set, and the supply nets they represent, can be connected to instances in one of the following ways: explicitly, automatically, or implicitly. Connections are made downward, from ports or nets in the current scope to ports of descendant instances that are in the extent of the domain.

4.4.2.5.1 Explicit and automatic connections

An explicit connection connects a given particular supply set function directly to a specified supply port. See also 6.11 and 6.12.

An automatic connection connects each supply set function to ports of selected instances, based on the *pg_type* of each port, as indicated by the **UPF_pg_type** attribute (see 6.46) or the Liberty `pg_type` attribute.

For automatic connections, the default connection semantics for each function of a supply set are as follows:

a) **power** is connected by default to ports having the *pg_type* `primary_power`.

b) **ground** is connected by default to ports having the *pg_type* `primary_ground`.

c) **pwell** is connected by default to ports having the *pg_type* `pwell`.

d) **nwell** is connected by default to ports having the *pg_type* `nwell`.

e) **deeppwell** is connected by default to ports having the *pg_type* `deeppwell`.

f) **deepnwell** is connected by default to ports having the *pg_type* `deepnwell`.

4.4.2.5.2 Implicit connections

An implicit connection connects the power and ground functions of a supply set to cell instances that do not have explicit supply ports. Such connections may involve implicit creation of ports and nets, as described in 4.2.9.

Implicit supply set connections are made in each of the following cases:

a) Primary supply set

The functions of a domain's primary supply set are implicitly connected to any instance in the extent of the domain if the instance has no supply ports defined on its interface.

b) Retention supply set

The functions of a retention strategy's supply set are implicitly connected to the state element that implements retention functionality (e.g., a balloon latch, shadow register, or live slave latch) for any register in the domain to which the strategy applies.

c) Isolation supply set

The functions of a supply set for an isolation strategy are implicitly connected to the corresponding isolation cell implied by the application of the strategy.

d) Level-shifter supply sets

The functions of a supply set for a level-shifting strategy are implicitly connected as appropriate to the input, output, or internal supply pins of any level-shifter implied by the application of the strategy.

After UPF-specified power intent has been completely applied, it shall be an error if any instance in the design does not have a supply set function or supply net connected to each of its supply ports, including any implicit power and ground ports.

4.4.2.6 Supply set required functions

Although a supply set represents a collection of six standard supply functions, not all functions are required in every context:

— `power` and `ground` are typically required in all cases.

— `nwell`, `pwell`, `deepnwell`, and `deeppwell` are only required occasionally.

The required functions of a given supply set are determined from its usage and include the following:

a) Any function used to define a power state of the supply set,

b) Any function used for automatic connection of the supply set based on *pg_type,* and

c) Any required function of a supply set handle with which the supply set is associated.

For implementation, a supply net shall be associated with each required function of a supply set. For verification, however, some aspects of the power intent can be verified before associating supply nets with the required functions. A supply set that does not have supply nets associated with each of its required functions is incompletely specified. For any required function of a supply set that is not associated with a supply net, an implicit supply net is created and associated with the function.

4.4.3 Supply equivalence

Various aspects of power management are determined in part by the identify of, and relationships between, supply nets and supply sets. For example, selection of ports to which isolation or level-shifting strategies should be applied can be defined based on the identities of the driver and receiver supplies of the sources and sinks connected to a port. Similarly, composition of power domains is possible provided the supplies of the subdomains involved meet certain constraints. In some situations, identical supply nets or supply sets are required; other situations may only require supply nets or supply sets that are equivalent.

There are two kinds of supply equivalence: electrical equivalence and functional equivalence.

Electrical equivalence can affect

a) the number of sources of a supply network, and therefore,

b) whether resolution is required for that supply network.

Electrical equivalence implies functional equivalence, but not vice versa.

Functional equivalence can affect any of the following:

c) Insertion of isolation cells, level-shifter cells, and repeater cells

d) Determination of power-domain lower boundaries

e) Legality of power-domain composition

f) Validity of driver and receiver supply attributes

Electrical equivalence is primarily related to supply ports and nets. Functional equivalence is primarily related to supply sets.

4.4.3.1 Supply port/net equivalence

Electrical equivalence is determined by connection, as follows:

a) A port `P` is electrically equivalent to itself.

b) A net `N` is electrically equivalent to itself.

c) If a net `N` and a port `P` are connected, then `N` and `P` are electrically equivalent.

d) If `A` and `B` are electrically equivalent, and `B` and `C` are electrically equivalent, then `A` and `C` are electrically equivalent.

e) If `A` and `B` are connected via a supply set function (see 4.4.2.4), then `A` and `B` are electrically equivalent.

Electrical equivalence can also be declared, as follows:

— If `A` and `B` are declared electrically equivalent, then `A` and `B` are electrically equivalent.

Electrical equivalence implies the two equivalent objects are electrically connected somewhere. If the connection is not evident in the design (e.g., if it is inside a hard macro whose internals are not visible or if it is a connection that is required outside the design), then declaration of electrical equivalence can be used instead of the explicit connection.

Functional equivalence is determined by connection or declaration, as follows:

f) If `A` and `B` are electrically equivalent, then `A` and `B` are functionally equivalent.

g) If `A` and `B` are declared functionally equivalent, then `A` and `B` are functionally equivalent.

An input and the output of a switch are never electrically equivalent; it is an error if they are directly connected or declared electrically equivalent. Similarly, the outputs of two different switches are typically not electrically equivalent, unless they are both driving the same resolved net. However, the outputs of two different switches that each drive an unresolved net can still be functionally equivalent if the input supplies of both switches are equivalent, the control inputs of both switches are equivalent, and the two switches have the same set of state definitions.

4.4.3.2 Supply set equivalence

A supply set handle is also a supply set.

A supply set function and its associated supply net are electrically equivalent; thus, for purposes of supply net equivalence, a supply set function acts like a supply net.

Corresponding functions of two supply sets are electrically equivalent if

— their associated supply nets are electrically equivalent, or

— the two supply sets are directly associated with one another.

Corresponding functions of two supply sets are functionally equivalent if

— they are electrically equivalent, or

— they have been declared as functionally equivalent.

Two supply sets are (functionally) equivalent if

— they both have the same required functions, and the nets associated with corresponding functions are equivalent; or

— they are associated with each other directly or indirectly via one or more **associate_supply_set** commands (see 6.7); or

— they are each associated directly or indirectly via **associate_supply_set** (see 6.7) with two other supply sets, which are equivalent.

Two supply sets are also (functionally) equivalent if they have been declared equivalent; in this case, it is an error if they do not have the same required functions.

As a consequence of this,

a) two anonymous supply sets built from equivalent PG functions are equivalent;

b) two supply sets that are functionally equivalent can be used interchangeably;

c) a supply set and any supply set handle it is associated with are always equivalent.

4.5 Power management

While a power supply network is a static structure, the power delivered via the power supply network can vary over time. Supply sources can provide different voltages; power switches can turn their outputs off or on and can selectively connect different inputs to the output. As a result, the power available to instances in the extent of a power domain will vary, and at any given time, each power domain's supplies may be in one of many possible states. To manage these various states, and in particular to manage the interactions between power domains that are in different states, power management is required.

Power management enables a system to operate correctly in a given functional mode with the minimum power consumption. Adding power management to a design involves analyzing the design to determine which power supplies provide power to each logic element, and if the driver and receiver are in different power domains, inserting power-management cells as required to ensure that neither logical nor electrical problems result if the two power domains are in different power states.

4.5.1 Related supplies

An active component consists of logic elements that receive inputs and drive outputs. The power supplies connected to an active component provide power for this logic. The supply nets that provide power for the logic that receives or drives a given input or output, respectively, are called the *related supplies* of that input or output. Related supplies typically include power and ground supplies and may also include bias supplies.

At the library cell level, related supplies may be identified for each input or output pin of a cell. Each related supply is a supply pin on that cell; the pin typically has a `pg_type` attribute indicating what supply function it provides (primary power, primary ground, etc.). For a cell that has one set of supply connections, all inputs and outputs would have the same set of related supplies. For a cell that has multiple supply connections, such as a cell with a backup power supply, different pins may have different sets of related supplies. This is particularly true of certain power-management cells, such as a level-shifter, which usually has different related supplies for the input and output.

Related supply nets are often considered in a group, as an implicit supply set. An implicit supply set made up of the supply pins of a cell that are the related supplies of a given input or output is by definition equivalent to any supply set that has been connected to those supply pins.

4.5.2 Driver and receiver supplies

Each output of an active component is typically connected to the input of some other active component in the design. The net connecting the two has a driver on one end (the logic driving the output port) and a receiver on the other end (the logic receiving the input). The driving logic is powered by a supply set called the *driver supply*; the receiving logic is powered by a supply set called the *receiver supply*.

The driver supply and the receiver supply may be the same supply set, e.g., if both components are in the same power domain; or the driver supply and the receiver supply may be different supply sets, e.g., if the two components are in different power domains. The driver supply and the receiver supply may also be

different, but nonetheless equivalent, e.g., if they are connected externally or if they are generated by supply networks that ensure they always have the same values.

In some cases, the logic driving or receiving a given port is not evident. In particular, the logic inside a hard macro instance may not be represented in a way that can be used by a given tool. Similarly, the logic that drives primary inputs of the design and receives primary outputs of the design is typically not represented as part of the design. In such cases, it is convenient to be able to associate the driver supply or receiver supply of the missing logic with the port that is connected to that logic. UPF defines attributes that can be used to associate this information with ports of a model.

4.5.3 Logic sources and sinks

Logic ports can be a source, a sink, or both, as follows:

- The LowConn of an input or inout logic port whose HighConn is connected to an external driver is a source.
- The HighConn of an output or inout logic port whose LowConn is connected to an internal driver is a source.
- The LowConn of an output or inout logic port whose Highconn is connected to an external receiver is a sink.
- The Highconn of an input or inout logic port whose LowConn is connected to an internal receiver is a sink.

For a logic port that is connected to a driver, the supply of the connected driver is also the driver supply of the port. A primary input port is assumed to have an external driver and therefore is a source; such a port has a default driver supply if it does not have an explicitly defined **UPF_driver_supply** attribute. An internal port that is not connected to a driver is not a source, and therefore, does not have a driver supply in the design. To model this in verification, an anonymous default driver is created for such an undriven port. This driver always drives the otherwise undriven port in a manner that results in a corrupted value on the port.

For a logic port that is connected to one or more receivers, the supplies of the connected receivers are all receiver supplies of the port. A primary output port is assumed to have an external receiver and therefore is a sink; such a port has a default receiver supply if it does not have an explicitly defined **UPF_receiver_supply** attribute. An internal port that is not connected to a receiver is not a sink, and therefore, does not have any receiver supplies.

4.5.4 Power-management requirements

Power management is required to mediate the changing power states of power domains in the system and the interactions between power domains that are in different states at various times. There are four specific areas addressed by power management, as follows:

- If a power domain is powered down in certain situations, its state registers may need to have their values saved before power-down and restored after subsequent power-up, either to maintain persistent data or to enable faster power-up.
- If the distance between driver and receiver is long (the capacitive load is high), buffers (repeaters) may be required to strengthen the signal along the way, or to ensure that it stabilizes within the required time.
- If a receiver is powered on, but its driver is not, an isolation cell is required between driver and receiver to drive the receiver with a known value despite the fact that the ultimate driver is powered off.
- If the driver and receiver supplies (or isolation and receiver supplies, or driver and isolation supplies, etc.) are operating at different voltage levels, a level-shifter is required between them to translate between voltage levels.

UPF provides commands for specifying where power-management structures should be added to a design to address each of these areas.

4.5.5 Power-management strategies

Addition of power-management cells to a design is driven by rules or strategies. UPF provides commands for specifying retention strategies (see 6.49), repeater strategies (see 6.48), isolation strategies (see 6.41), and level-shifting strategies (see 6.43). Each of these strategies can be defined in various ways to apply to specific design features or more generally to classes of features. Precedence rules (see 5.8) define how multiple strategies for the same feature are to be interpreted. In general, more specific strategies take precedence over more general strategies.

Retention strategies apply to specific state variables in a given power domain or to all state variables in a domain. A retention strategy also defines the power supplies, the control signals and their interpretation, and certain behavioral characteristics of the retention registers to be used for the state variables to which it applies.

Repeater, isolation, and level-shifting strategies apply to ports of a power domain. The ports to which one of these strategies applies can be defined by name or can be selected by filters. Source and sink filters select ports based on the driver supply and receiver supply, respectively, of each port. The filters typically match equivalent supplies unless an exact match is specified. Ports may also be selected by direction. Each of these strategies also specifies the relevant power supplies and control signals and their interpretation to be used for any power-management cells added by the strategy.

4.5.6 Power-management implementation

Implementation of power-management strategies involves adding power-management cells—retention registers, repeaters (buffers), isolation cells, and level-shifter cells—to the design. Each added cell may add new driving and receiving logic and as a result may change the driver and receiver supplies of a given port, which could potentially affect the application of other strategies based on source and sink filters. To ensure the interaction of multiple strategies is well defined, strategies are applied according to the following rules.

a) Strategies are implemented in the following order: retention strategies, followed by repeater strategies, followed by isolation strategies, followed by level-shifter strategies.

b) A retention strategy may affect the driving supply of the retention cell output. If so, the new driving supply of the retention cell is visible to, and affects the result of, a source filter of any subsequently applied strategy.

c) A repeater strategy causes insertion of a buffer, which has a receiver and a driver; this insertion therefore affects both the receiving supply of ports driving the repeater input and the driving supply of ports receiving the repeater output. The new driving supply and receiver supply are visible to, and affect the result of, source and sink filters, respectively, of any subsequently applied strategy.

d) An isolation strategy may cause insertion of an isolation cell, which has a receiver and a driver; therefore if such insertion occurs, it affects both the receiving supply of ports driving the isolation cell input and the driving supply of ports receiving the isolation cell output. However, the new driving supply and receiver supply are not visible to, and do not affect the result of, source and sink filters, respectively, of any subsequently applied isolation or level-shifting strategies.

e) A level-shifting strategy may cause insertion of a level-shifting cell, which has a receiver and a driver; therefore if such insertion occurs, it affects both the receiving supply of ports driving the level-shifting cell input and the driving supply of ports receiving the level-shifting cell output. However, the new driving supply and receiver supply are not visible to, and do not affect the result of, source and sink filters, respectively, of any subsequently applied level-shifting strategy.

Repeater, isolation, and level-shifting strategies apply to all ports on the interface of a power domain, both those on the upper boundary of the domain and those on the lower boundary of a domain. As a result, a port on the boundary between two domains—the upper boundary of one, and the lower boundary of the other—may have multiple strategies of a given type defined for it, one from each of the two domains. In such a case, both strategies may cause addition of power-management cells.

4.5.7 Power control logic

Power-management elements require control signals to coordinate their activity. In particular, isolation cells require enable signals, retention cells may require save and restore signals or related control inputs, and power switches (see 4.4.1.2) require switch control signals. Logic ports and nets that implement these control signals may be present already in the HDL design or they may be added via UPF commands.

Control logic ports and nets defined in UPF are created within the logic hierarchy independent of power-domain definitions. This allows the power control network to be created and distributed across power domains.

4.6 Power states

Supply ports, supply nets, supply sets, and power domains have associated power states. The power state of a supply port or net at a given time is the value propagated by that port or net. For a supply set or power domain, power states are defined based on supply port/net power states and other conditions.

Power switches also have named states. These are not power states of the switch, but rather states of the control expressions that determine which inputs of a switch affect the switch output (see 4.4.1.2).

4.6.1 Power state of a supply port or supply net

Supply ports and nets are represented by type **supply_net_type**, defined in package UPF (see Annex B). This type models electrical values as a combination of two values: a supply state and a voltage level, which together constitute the power state of the supply port or net.

The supply state value may be **OFF**, **UNDETERMINED**, **PARTIAL_ON**, or **FULL_ON**. The supply state value is not affected by or determined by the supply voltage level.

The voltage level is represented as an integer number of microvolts. The voltage level is relevant only for the **PARTIAL_ON** and **FULL_ON** supply states; it is undefined for the **OFF** and **UNDETERMINED** supply states.

4.6.2 Power state of a supply set

A supply set consists of a collection of functions that represent supply nets. A supply set has a reference supply net. The default reference is an implicit supply net with a supply state of **FULL_ON** and a voltage value of zero. The default reference supply can be explicitly overridden by specifying a supply net that is used as the reference supply for every supply net in the set. The voltage value of each supply net in a supply set is relative to the reference supply, which, in turn, may be at any voltage relative to the implicit reference supply.

Power states of a supply set are defined in terms of the power states of the supply functions that comprise the supply set, and the supply nets those functions represent, as well as related control conditions. The combined states of the constituent supply functions/nets and control conditions determine the following:

— Whether there is current available to power an instance, and
— The voltage level of the supply.

Power state definitions for a supply set are predicates: each one defines a set of conditions that, if satisfied, indicates that the supply set is in the corresponding state. Power state definitions need not be mutually exclusive; multiple power state definitions can be satisfied at any given time.

A supply set handle is also a supply set. Power states may be defined for a supply set handle as well as for a supply set.

Power state definitions for supply sets and supply set handles are only associated with and only apply to the supply set or supply set handle for which they are explicitly defined. They do not propagate to or apply to other associated supply sets or supply set handles.

4.6.3 Predefined supply set power states

Every supply set has two predefined power states: **DEFAULT_NORMAL** and **DEFAULT_CORRUPT**. These power states are identical to explicitly defined power states except: It is an error if **DEFAULT_NORMAL** and **DEFAULT_CORRUPT** are used as the *state_name* in an **add_power_state** command (see 6.4).

A supply set is in the **DEFAULT_NORMAL** state when all of its required supply functions are **FULL_ON**.

The supply set is in the **DEFAULT_CORRUPT** power state when it is not in one of the defined power states of the supply set, including the **DEFAULT_NORMAL** predefined state, for the supply set.

4.6.4 Power states of power domains

A power domain typically represents a collection of instances that are powered with the same supplies. Power states of such a domain can be defined in terms of the power states of supply sets associated with the domain. Such definitions implicitly act as constraints on the power supplies provided to the domain.

A power domain may also be used to represent the interface to an IP block, which may contain multiple power domains. Power states of such a domain can be defined in terms of power states of the other domains in the IP block. Such definitions typically represent abstract power states of any given instance of the IP block.

For example, the definition of a domain's `POWER_ON` power state would logically require the primary supply set be in a power state in which all supply nets of the primary supply set are on and the current delivered by the power circuit is sufficient to support normal operation. Similarly, a `SLEEP` power state for the domain may require the primary supply set to be in power state in which sufficient voltage and current is provided to maintain the state of registers, but not enough to support normal operation. A `POWER_OFF` power state may require the primary supply set to be switched off, while the appropriate retention and isolation supplies are on.

The state of logic elements may be a relevant aspect to the specification of a domain's power state, e.g., for a user-defined power domain called `DSP_PD`,

a) `DSP_PD` is in the state `my_on_pd_state` when:
   1) The logic signal that controls the switch for the domain's primary supply set is active.
   2) The logic signal(s) enabling isolation are inactive.
b) `DSP_PD` is in the state `my_off_pd_state` when:
   1) The logic signal that controls the switch for the primary supply is inactive.
   2) If the isolation or retention supplies are switched, the control signals for those supplies are active (the power switch is on).
   3) Clock gating enable signals for the domain are typically inactive.

4) The isolation enable(s) are active.

5) The retention control signal(s) are active.

c) The power domain's power state may also be dependent on the clock period or similar signal interval constraint. For example, a domain in an operational bias mode needs to scale its clock frequency to a slower level to match the slower switching performance supported by the state of the primary supply set. The primary supply set's power state can include in the **-logic_expr** specification a constraint on the clock period or duty cycle interval. See 6.4.

4.6.5 Power states of systems and subsystems

What constitutes a system is contextual. In one context, a system may be considered as complete by itself, e.g., one chip of a multi-chip or multi-board low-power system. Although it might seem reasonable to define a "system" as that which is automatically implemented, UPF is not limited to that context and the verification of an entire system composed of multiple chips each with its own power intent specification, as well as an overall power intent specification for the board on which the chips are placed, is supported. The power states of a system or subsystem are attributed on a power domain. The use of the term *system* includes the term *subsystem*.

As a system power state may depend on the state of more than one power domain, the power state specification for a power domain may include references to the states of domains defined on scopes in the logic hierarchy that are descendants of the "higher-level" power domain. (Here, "higher-level" refers to the location of the power domain's scope being closer to the design's top-level root instance relative to the scope of another power domain.) Therefore, UPF allows a power state definition of a given power domain to reference the power state of any power domain or supply set, or the port state of any supply port or supply net, that is declared in the descendant subtree of the scope of that power domain.

For example, assume the domain `CORE_PD` is defined on the root scope of a processor design, the power states of `CORE_PD` can reference lower-level power domains such as `CACHE_PD`, `ALU_PD`, and `FP_PD` in the specification of its power states. Thus, an example power state of `FULL_OP` for `CORE_PD` would reasonably require that its primary supply set is in a **NORMAL** simstate (all supply nets of the primary supply set are on and the voltage of the supply is sufficient for normal operations) and that the `CACHE_PD`, `ALU_PD`, and `FP_PD` are all in an equivalent fully operational mode. In contrast, a `NON_FP_OP` mode for the `CORE_PD` may be defined identically to `FULL_OP`, except the `FP_PD` may be in a `SLEEP` mode. By allowing a higher level domain to reference lower-level domain's power states in the specification of its own power states, subsystem and system power states can be defined in UPF.

NOTE—Although the top-down specification of power states suggests a power domain's power states are defined in terms of the power states of supply sets and lower-level power domains, the power state of a domain can be specified entirely in terms of the state of supply sets and/or supply nets and supply ports; i.e., the hierarchical specification can be collapsed into a (relatively) flat power state specification. Top-down, hierarchical power state specification is convenient when the power design starts prior to the existence of the complete supply network and is refined into an implementation. The flat specification of power states of domains in terms of direct references to supply nets may be faster and more concise when the power state specification is not captured until after the supply network is specified. However, flat power state specifications may be less flexible and more difficult to maintain over time and require visibility into and understanding of all aspects of the design.

4.6.6 Incremental refinement of power states

Prior to having the golden source (the HDL and UPF source used as input to implementation tools), the supply network may not be defined or may be partially defined. The design may have a power-management block and associated power control signals that turn power switches on/off or control bias generators and voltage regulators once the supply network is fully specified. At this stage of design specification, the power domain's power states may be defined only in terms of the state of logic elements, i.e., control signals.

Power states of the domain's supply set handles may be added later as the supply network definition is completed. Through support of incremental refinement of the power state specification, early UPF simulations can be performed with only the logic net expressions defining the states. The power state definitions can be updated with **add_power_state** (see 6.4) to incorporate supply network expressions (**-supply_expr**) or additional logic expressions (**-logic_expr**).

4.7 Simstates

Simstates specify the simulation behavior semantics for a power state. A simstate specifies the level of operational capability supported by a supply set state. The simstate specification provides digital-simulation tools sufficient information for approximating the power-related behavior of logic connected to the supply set with sufficient accuracy.

Simstates are associated with power states of supply sets and supply set handles. A simstate defines how instances powered by the supply set or supply set handle react to a given power state. In particular, simstates can be associated with power states of the primary supply of a power domain, to define how instances in the power domain that are implicitly connected to that primary supply will behave under various power states of the primary supply.

UPF defines several simstates that can be associated with supply set or supply set handle power states. The simstates defined in UPF are an abstraction suitable for digital simulation. The following simstates are defined (from highest to lowest precedence):

a) **CORRUPT**—The supply set is either off (one or more supply nets in the set are switched off, terminating the flow of current) or at such a low-voltage level that it cannot support switching and the retention of the state of logic nets cannot be guaranteed to be maintained even in the absence of activity in the instances powered by the supply.

b) **CORRUPT_ON_ACTIVITY**—The power characteristics of the supply set are sufficient for logic nets to retain their state as long as there is no activity within the elements connected to the supply, but they are insufficient to support activity.

c) **CORRUPT_ON_CHANGE**—The power characteristics of the supply set are sufficient for logic nets to retain their state as long as there is no change in the outputs of the elements connected to that supply.

d) **CORRUPT_STATE_ON_ACTIVITY**—The power characteristics of the supply set are sufficient to support normal operation of combinational logic, but they are insufficient to support activity inside state elements, whether that activity would result in any state change or not.

e) **CORRUPT_STATE_ON_CHANGE**—The power characteristics of the supply set are sufficient to support normal operation of combinational logic, and they are sufficient to support activity inside state elements, but they are insufficient to support a change of state for state elements.

f) **NORMAL**—The power characteristics of the supply set are sufficient to support full and complete operational (switching) capabilities with characterized timing.

The predefined power states for a supply set have corresponding simstates. The simstate for power state **DEFAULT_NORMAL** is **NORMAL**. The simstate for power state **DEFAULT_CORRUPT** is **CORRUPT**.

Simstate simulation semantics for a supply set are applied to instances implicitly connected to a supply set unless simstate behavior has been disabled (see 6.53).

NOTE 1—When greater accuracy is desired or required, a mixed signal or full analog simulation can be used. Since analog simulations already incorporate power, this format provides no additional semantics for analog verification.

Simulation results reflect the implemented hardware results only to the extent the UPF simstate specification for a given power state of a supply set is correctly specified. For example, if verification is performed with simulation of a supply set in a power state specified as having a **CORRUPT_ON_ACTIVITY** simstate, but the implementation is more accurately classified as **CORRUPT_STATE_ON_CHANGE**, the simulation results will differ.

NOTE 2—In this example, the inaccuracy in simstate specification is conservative relative to the implemented hardware behavior. However, in other situations, inaccurate specifications can be optimistic, resulting in errors in the implemented hardware that simulation failed to expose.

4.8 Successive refinement

Design and implementation of a power-managed system using UPF proceeds in stages. During the design phase, a UPF-based specification of the power intent may be developed incrementally, first at the IP block level, and later at the system level. During implementation, UPF commands are added to drive implementation details, and a series of implementation steps map the design and the UPF commands into the final implementation (see Figure 2).

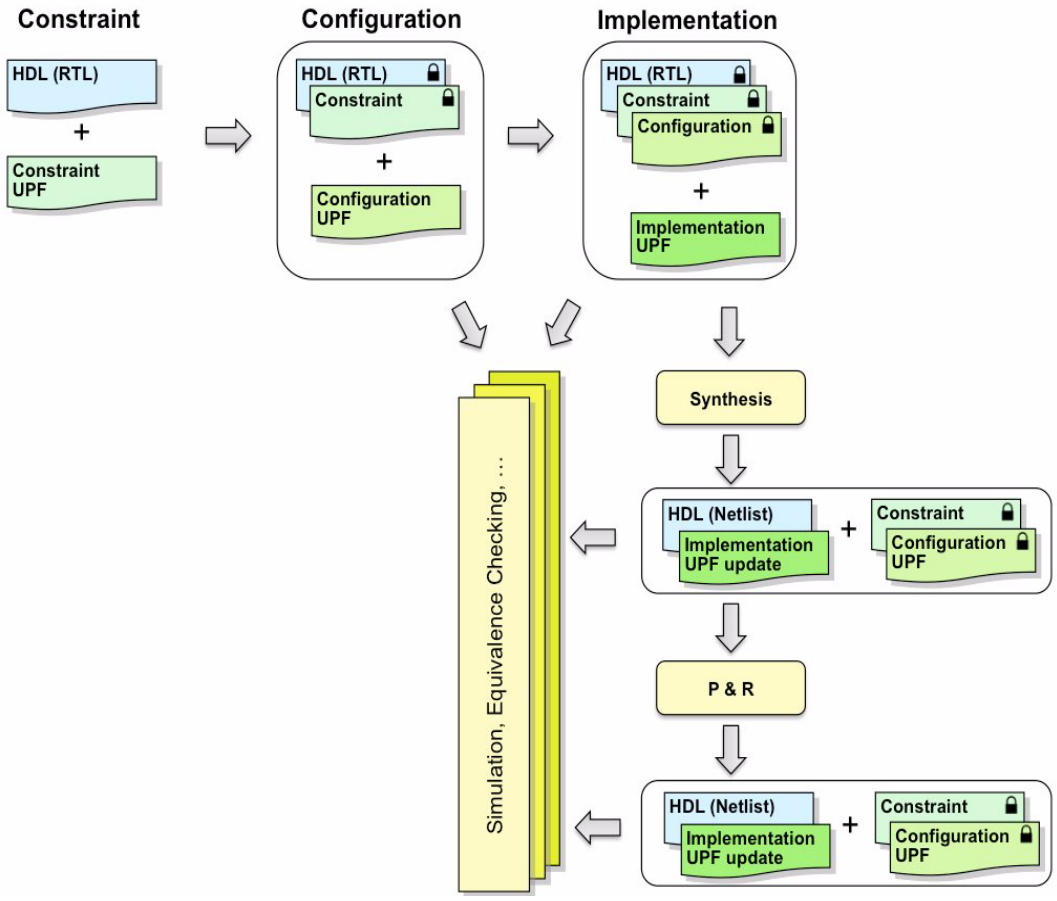

Figure 2—Successive refinement of power intent

The power intent specification for an IP block to be used in a larger design typically defines the power interface to the block and the power domains within the block. This specification also typically includes constraints on the use of the block in a power-managed environment. These constraints include (at least) the following:

a) The atomic power domains in the design.

   These can be composed but not split during implementation. [Use **create_power_domain –atomic** (see 6.17).]

b) The state variables that need to have their values retained if a given power domain is powered down.

   This does not involve specifying how such retention would be controlled. [Use **set_retention_elements** (see 6.51).]

c) The clamp values of signals that would need to be isolated if a given power domain is powered down.

   This does not involve specifying how isolation is to be controlled. [Use **set_port_attributes –clamp_value** (see 6.46).]

d) The legal power states and power state transitions of the IP block's power domains.

   This need not involve specifying absolute voltage ranges for the power supplies involved. [Use **add_power_state** (see 6.4) and **describe_state_transition** (see 6.24).]

A power intent specification containing such basic information about an IP block is often referred to as *constraint UPF*, or sometimes as the *platinum UPF*.

When an IP block is being prepared for use in a given system, information may be added to the specification to reflect the specific requirements of the block in the context of the system. For example, an instance of the block may be used in a manner that will definitely require isolation, level-shifting, retention, or repeater cell insertion. These strategies can be added to the constraint UPF for the block in order to configure the power intent of the block for use in this system. Such strategies impose a requirement to insert specified power-management cells for an instance of the IP block and typically include information about how such power-management cells are controlled.

A power intent specification containing this level of information is often referred to as *configuration UPF*, or sometimes as the *golden UPF*.

To drive implementation of a power-managed design, information may be added to the specification to define the power distribution network for the system and the control logic for power-management cells. A power intent specification containing this kind of information is often referred to as *implementation UPF*, or sometimes as the *silicon UPF*.

4.9 Tool flow

A UPF-based tool flow typically begins with RTL verification of the design together with the golden UPF that defines the power intent for this design. After that, a series of implementation steps occur in which the RTL design is reduced to a gate-level implementation and the power intent is integrated into that implementation. After each implementation step, power-aware verification may be performed again, using the design representation output by that stage along with the UPF description corresponding to that design representation (see Figure 2).

The power intent expressed in UPF can be implemented incrementally in successive steps. Each step may add implementation details, such as power-management cells, control logic, or supply distribution networks. The design itself may also evolve during implementation, even after the RTL stage, as a result of implementation steps such as test insertion.

Implementation may be incremental at various levels of granularity as follows:

— By aspect: isolation, level-shifting, retention, repeaters, control logic, power distribution
— By command: isolation strategy A, isolation strategy B, etc.
— By element to which a command applies: isolation for port `p1`, for port `p2`, etc.

For any given tool run, the tool needs to know the following:

a) What part of the UPF power intent specification is supposed to be implemented already, and
b) What part of the UPF power intent specification is to be included in the processing done by this tool.

This standard does not define how the preceding information is made available to a tool; this is tool/flow information that is outside the scope of the standard. Typically, such information would be provided to the tool either explicitly via command-line arguments or other control inputs, or implicitly as part of the specification of the tool itself.

A tool also shall be able to determine what part of the UPF specification has been implemented so far. This standard defines a method for documenting what has been done so far to implement the power intent, by identifying ports, nets, and instances in the design that represent implementations of UPF commands.

4.10 File structure

For maximum reuse, it may be appropriate to keep constraint, configuration, and implementation UPF commands in separate files. The **load_upf** command (see 6.28) can be used to compose the files for a particular context.

For example, an IP block with a corresponding constraint UPF description might be configured for use in a given system by creating a configuration UPF file for it. The configuration UPF file would load the constraint UPF for the IP block and then continue with additional commands defining or updating the isolation, level-shifting, retention, and repeater strategies required for this configuration of the IP block. Different configuration UPF files can be constructed based on the same constraint UPF, to define different configurations of the same IP block for use in different situations.

For implementation of the design, an implementation UPF file may be constructed by loading the configuration UPF for the various IP blocks involved in the system and then adding implementation details, such as supply ports, nets, and sets, power switches, port attributes, and supply connections. Different implementation UPF files can be constructed using the same configuration UPF files, to evaluate or verify alternative implementations.

For each implementation step, tools may update the implementation UPF to document the additions made to the design in that step to implement the power intent. To keep the implementation updates separate from the input UPF specification, a tool may generate an output UPF file that loads the input UPF file and then adds UPF command updates as required. Successive implementation steps may choose to append to this update file or generate a new update file that loads the previous one.

5. Language basics

5.1 UPF is Tcl

UPF is based on Tool Command Language (Tcl). UPF commands are defined using syntax that is consistent with Tcl, such that a standard Tcl interpreter can be used to read and process UPF commands.

Compliant processors reading UPF files use full Tcl interpreters to process the UPF files. Compliant processors shall use Tcl version 8.4 or above. The following also apply:

— UPF power intent commands are executed in the order of occurrence, just as Tcl commands are executed and return values can be used by subsequent commands.

— The only UPF commands that support regular expressions are **find_objects** (see 6.26) and **query_upf** (see C.1).

— All of the commands and techniques of Tcl may be used, including procs and libraries of procs. However, the procs and libraries of procs should, in the end, only rely on UPF commands for design information.

— **find_objects** (see 6.26) shall be the only source used to programmatically access the HDL when defining the power intent. The processing of information returned by **find_objects** using standard Tcl commands [B5], such as `regexp`, is allowed.

— UPF is intended to be used across many tools, so it is erroneous to use proprietary tool specific commands when constructing power intent.

— Once the Tcl processing has completed, the end result can be expressed as a series of UPF commands.

Libraries used for design or methodology standardization or ease of expression that define additional procs are considered to be part of the design file and need to be visible to any processor interpreting the UPF file.

5.2 Conventions used

Each UPF command in Clause 6 and Clause 7 consists of a command keyword followed by one or more parameters. All parameters begin with a hyphen (-). The meta-syntax for the description of the syntax rules uses the conventions shown in Table 1.

Table 1—Document conventions

Visual cue	Represents
`courier`	The `courier` font indicates UPF or HDL code. For example, the following line indicates UPF code: `create_power_domain PD1`
bold	The **bold** font is used to indicate key terms, text that shall be typed exactly as it appears. For example, in the following command, the keywords “create_power_domain” shall be typed as it appears: **create_power_domain** *domain_name*
italic	The *italic* font represents user-defined UPF variables. For example, a supply net shall be specified in the following line (after the “connect_supply_net” key term): **connect_supply_net** *net_name*

Table 1—Document conventions *(continued)*

Visual cue	Represents
list	*list* (or *xyz_list*) indicates a Tcl list, which is denoted with curly braces **{**….**}** or as a double-quoted string of elements **"**….**"**. When a list contains only one non-list element (without special characters), the curly braces can be omitted, e.g., `{a}`, `"a"`, and `a` are acceptable values for a single element. See also 5.3.4.
xyz_ref	*xyz_ref* can be used when a symbolic name (i.e., using a handle) is allowed as well as a declared name, e.g., *supply_set_ref*.
time_literal	*time_literal* indicates a SystemVerilog or VHDL `time_literal`.
* asterisk	An asterisk (`*`) signifies that a parameter can be repeated. For example, the following line means multiple acknowledge delays can be specified for this command: [**-ack_delay** {*port_name delay*}]*
[ ] square brackets	Square brackets indicate optional parameters. If an asterisk (`*`) follows the closing bracket, the bracketed parameter may be repeated. For example, the following parameter is optional: [**-elements** *element_list*] The following is an example of optional parameter that can be repeated: [**-ack_port** {*port_name net_name* [{*boolean_expression*}]}]*
[ ] bold square brackets	Bold square brackets are required. For example, in the following parameter, the bold square brackets (surrounding the `0`) need to be typed as they appear: *domain_name*.*isolation_name.isolation_supply_set***[0]**
{ } curly braces	Curly braces (`{ }`) indicate a parameter list that is required. In some (or even many) cases, they have (or are followed by) an asterisk (`*`), which indicates that they can be repeated. For example, the following shows one or more control ports can be specified for this command: {**-control_port** {*port_name*}}*
{ } bold curly braces	Bold curly braces are required, unless the argument is already a Tcl list. For example, in the following parameter, the bold curly braces need to be typed as they appear: [**-off_state** {*state_name* {*boolean_expression*}}]* In cases where variable substitution is needed, Tcl's list command can be used, e.g., `-off_state [list $state_name [list $expression]]`
< > angle brackets	Angle brackets (`< >`) indicate a grouping, usually of alternative parameters. For example, the following line shows the "power" or "ground" key terms are possible values for the "-type" parameter: **-type** <**power** | **ground**>
| separator bar	The separator bar (`|`) character indicates alternative choices. For example, the following line shows the "in" or "out" key terms are possible values for the "-direction" parameter: **-direction** <**in** | **out**>

5.3 Lexical elements

Names created in UPF should not conflict with HDL reserved words.

Command names, parameter names, and their values are case-sensitive.

5.3.1 Identifiers

Identifiers adhere to the following rules:

a) The first character of a identifier shall be alphabetic.
b) All other characters of a identifier shall be alphanumeric or the underscore character (_).
c) Identifiers in UPF are case-sensitive.

5.3.2 Keywords and reserved words

The following record field names are reserved in the specified context and cannot be redefined:

a) Domain record field space
    1) **primary**
    2) **default_retention**
    3) **default_isolation**
b) Switch record field space
    1) **supply**
c) Level-shifter strategy record field name space
    1) **input_supply_set**
    2) **output_supply_set**
    3) **internal_supply_set**
d) Isolation strategy record field name space
    1) **isolation_supply_set**
    2) **isolation_signal**
e) Retention strategy record field name space (see 6.33)
    1) **retention_ref**
    2) **retention_supply_set**
    3) **primary_ref**
    4) **primary_supply_set**
    5) **save_signal**
    6) **restore_signal**
    7) **UPF_GENERIC_CLOCK**
    8) **UPF_GENERIC_DATA**
    9) **UPF_GENERIC_ASYNC_LOAD**
    10) **UPF_GENERIC_OUTPUT**

5.3.3 Names

Names identify objects in the design and in the power intent specification.

5.3.3.1 Simple names

A simple name is a single identifier. An identifier is used when creating a new object in a given scope; the identifier becomes the simple name of that object.

In a given scope, a given simple name may only be defined once, with a unique meaning; it is an error if two objects are declared in the same scope with the same simple name.

A simple name, optionally followed by an index or record field specification as appropriate for the type of an object in a given HDL context, is an object name. An object name can be used to refer to an existing object or part of an existing object that is declared in the current scope. Object names also refer to objects defined in UPF that do not exist in a scope of the hierarchy.

The simple name of an instance in a given scope is an instance name.

5.3.3.2 Dotted names

A dotted name is a compound name designating a UPF object. A dotted name is made up of simple names separated by `.` characters.

A dotted name is used to refer to a strategy associated with a power domain, a supply set associated with a strategy or a power domain, or a function of a supply set. A dotted name for a supply set associated with a strategy or domain is called a *supply set handle*. A dotted name for a supply set function is called a *supply net handle*.

— Power-domain strategy names

```
<domain name> . <strategy name>
```

— Supply set handles

```
<domain name> . <supply set name>
<domain name> . <strategy name> . <supply set name>
```

— Supply net handles

```
<supply set name> . <function name>
<domain name> . <supply set name> . <function name>
<domain name> . <strategy name> . <supply set name> . <function name>
```

A dotted name is also an object name.

5.3.3.3 Hierarchical names

A hierarchical name is a name that refers to an object declared in a non-local scope. A hierarchical name consists of an optional leading `/` character, followed by a series of one or more instance names, each followed by the hierarchy separator character `/`, followed by an object name.

A hierarchical name that starts with an instance name is a scope-relative hierarchical name. A scope-relative hierarchical name is interpreted relative to the current scope. The first instance name is the name of an instance in the current scope; each successive instance name is the name of an instance declared in the scope of the previous instance. The trailing object name is the simple name or dotted name of an object declared in the scope of the last instance. A scope-relative hierarchical name is also called a *rooted name*.

A hierarchical name that starts with a leading `/` character is a design-relative hierarchical name. A design-relative hierarchical name is interpreted relative to the current design top instance, by removing the leading `/` character and interpreting the remainder as a rooted name in the scope of the current design top instance.

5.3.3.4 Name references

Many command arguments require references to object names, such as the names of instances, ports, registers, nets, etc., in the design, or the names of power domains, strategies, supply sets, supply nets, etc., in the power intent. Unless otherwise specified or contextually restricted, an object name reference can be a simple name, a dotted name, or a hierarchical name. In particular, a supply set handle is a form of supply set name and a supply net handle is a form of supply net name. In the absence of any statement to the contrary, a supply set handle can be used wherever a supply set name may appear, and a supply net handle can be used wherever a supply net name may appear.

5.3.4 Lists and strings

A Tcl list is an ordered sequence of zero or more elements, where each element can itself be a list. In Tcl, a string can be thought of as a list of words.

Tcl strings can be specified in two different ways: by enclosing the words within double-quotes (“”) or between curly braces ({}). Upon finding a list of words within double-quotes, Tcl continues to parse the string, looking for variable (strings started with $), command (strings between square brackets []), and back-slash (strings contain \) substitutions. To use any of the special characters within design object names, first wrap them in curly braces ({}). Upon finding a list of words between curly braces, Tcl treats the list as a literal list of words, preventing further processing on the list before it is used.

Therefore, in the syntax for UPF, the construct **-option** xxx_list can be satisfied by any of the following, when no special characters are used in the object names:

```
-option foo
-option “foo”
-option “foo bar bat”
-option {foo}
-option {foo bar bat etc}
```

5.3.5 Special characters

Special lexical elements (see Table 2) can be used to delimit tokens in the syntax.

Table 2—Special characters

Type	Character
Logic hierarchy delimiter	/
Escape character	\ (only escapes the next character)
Bus delimiter, index operator, or within a regex	[]
Range separator (for bus ranges)	:
Record field delimiter	.

When Tcl special characters need to be used literally for design object names, always escape the special character or wrap the name with {}, even if a single value is used, to protect from Tcl interpretation, e.g., -elements [list foo {foo/bar} a\[0\]].

5.4 Boolean expressions

A Boolean expression may be used to define a control condition or a supply state. A Boolean expression may include references to the following.

a) VHDL names, values, and literals of the following types or any subtype thereof:

   std.Standard.Boolean

   std.Standard.Bit

   std.Standard.Real for voltage values

   std.Standard.Time for use with the interval function

`ieee.std_logic_1164.std_ulogic`

`ieee.UPF.state`

b) SystemVerilog names, values, and literals of the following types:

`reg`

`wire`

`Bit`

`Logic`

`time_literal` for use with the `interval` function

`real`, `shortreal` for voltage values

A VHDL or SystemVerilog name may also be the name of an element of any composite type object provided the element itself is of a supported type.

A Boolean expression may also contain special expression forms for referring to power states (see 6.4).

A name of an object referred to in a Boolean expression may be prefixed by a path name identifying the instance in the scope of which the name is declared. Any such path name is interpreted relative to the current scope when the command defining the expression is executed. If no path name prefix is present, the name shall refer to an object declared in the current scope.

In a Boolean expression used as a supply expression in the definition of a power state of a supply set (handle), the name of any function of that supply set (handle) may be referred to directly without a prefix, unless such a reference would be ambiguous.

In a Boolean expression used as a logic expression in the definition of a power state of a power domain, the name of any supply set handle associated with that power domain may be referred to directly without a prefix, unless such a reference would be ambiguous.

A Boolean expression may include the operators shown in Table 3, which map to their corresponding equivalents in SystemVerilog or VHDL, as appropriate for the objects involved in each subexpression.

Table 3—Boolean operators

Operator	SystemVerilog equivalent	VHDL equivalent	Meaning
!	!	`not`	Logical negation
~	~	`not`	Bit-wise negation
<	<	<	Less than
<=	<=	<=	Less than or equal
>	>	>	Greater than
>=	>=	>=	Greater than or equal
==	==	=	Equal
!=	!=	/=	Not equal
&	&	`and`	Bit-wise conjunction
^	^	`xor`	Bit-wise exclusive disjunction

Table 3—Boolean operators *(continued)*

Operator	SystemVerilog equivalent	VHDL equivalent	Meaning
|	|	or	Bit-wise disjunction
&&	&&	and	Logical conjunction
||	||	or	Logical disjunction

A Boolean expression shall be provided as a string, as indicated in the syntax for each command in which a Boolean expression can appear. Subexpressions may be grouped with parenthesis (`()`). Logical operators have lowest precedence; bit-wise operators have next higher precedence; relational operators have next higher precedence; negation operators have highest precedence.

A Boolean expression or subexpression is considered to evaluate to the logical value *True* if evaluation of the expression (according to the semantics of the VHDL or SystemVerilog operators and types involved, as appropriate) results in a bit or logic value of `1` or a Boolean value of *True*; otherwise it is considered to evaluate to the logical value *False*.

A Boolean expression may contain references to objects in different language contexts provided that any given subexpression that evaluates to a logical (*True*/*False*) value contains only references to one language context. Logical negation, conjunction, and disjunction of logical values shall be performed according to standard Boolean logic semantics and need not be implemented with language-specific operators.

A simple expression is a Boolean expression containing an optional negation operator (**!** or ~), followed by optional white space and a single object name.

Examples

```
{ top/sv_inst/ena == 1'b1 && top/vhdl_inst/ready == '0' }
{ supply1.state == FULL_ON && supply1.voltage > 0.8 }
{(top/sv/wall.supply[0] != FULL_ON) || (top/vhdl/battery.supply(1) ==
   UNDEFINED)}
```

5.5 Object declaration

All UPF commands are executed in the current scope, except as specifically noted.

As a result, most objects created by a UPF command are created in the current scope within the design; therefore, the names of those objects shall not conflict with a name that is already declared within the same scope.

Some UPF objects are implicitly created. *Implicitly created objects* result from implied or inferred semantics and are not the direct result of creating a named UPF object. For example, supply nets are routed throughout the extent of a power domain as needed to implement the implicit and automatic connection semantics. This routing results in the creation of implicit supply ports and supply nets. UPF automatically names implicitly created objects to avoid creating a name conflict. The **name_format** command (see 6.35) can be used to provide a template for some implicitly created objects (such as isolation). Supply nets may be implicitly created and connected to supply ports, and logic nets may be implicitly created and connected to logic ports (see 4.4.1.1).

UPF objects may have record fields. These records comprise a name and a set of zero or more values. Record field names are in a local name space of the UPF object, e.g., a power domain may have strategies and supply set handles. Strategies themselves may also have supply set handles.

The `.` character is the delimiter for the hierarchy of UPF record fields, e.g., `top/a/PDa.MY_SUPPLY_SET` refers to the supply set `MY_SUPPLY_SET` in power domain `PDa` in the logical scope `top/a`.

5.6 Attributes of objects

HDLs include a mechanism for specifying properties of objects. These properties are called *attributes*. Certain UPF properties can be annotated directly in HDL source descriptions using attributes. The semantic for properties specified using HDL attributes is the same as the corresponding behavior defined by the UPF command alternative (see Clause 6). Table 4 enumerates the HDL attributes defined for UPF-compliant implementations.

Table 4—Attribute and command correspondence

HDL attribute name	Attribute value specification	Equivalent UPF command arguments	See
UPF_clamp_value	<**"0"** | **"1"** | **"Z"** | **"latch"** | **"any"** | **"***value***"**>	**set_port_attributes -clamp_value**	6.46
UPF_sink_off_clamp_value	<**"0"** | **"1"** | **"Z"** | **"latch"** | **"any"** | **"***value***"**>	**set_port_attributes -sink_off_clamp_value**	6.46
UPF_source_off_clamp_value	<**"0"** | **"1"** | **"Z"** | **"latch"** | **"any"** | **"***value***"**>	**set_port_attributes -source_off_clamp_value**	6.46
UPF_pg_type	*pg_type_value* (see 4.4.2.5)	**set_port_attributes -pg_type**	6.46
UPF_related_power_port	*supply_port_name*	**set_port_attributes -related_power_port**	6.46
UPF_related_ground_port	*supply_port_name*	**set_port_attributes -related_ground_port**	6.46
UPF_related_bias_ports	*supply_port_name_list*	**set_port_attributes -related_bias_ports**	6.46
UPF_driver_supply	*supply_set_ref*	**set_port_attributes -driver_supply**	6.46
UPF_receiver_supply	*supply_set_ref*	**set_port_attributes -receiver_supply**	6.46
UPF_feedthrough	<**"TRUE"** | **"FALSE"**>	**set_port_attributes -feedthrough**	6.46
UPF_unconnected	<**"TRUE"** | **"FALSE"**>	**set_port_attributes -unconnected**	6.46
UPF_retention	<**"required"** | **"optional"**>	**set_retention_elements -retention_purpose**	6.51
UPF_simstate_behavior	<**"ENABLE"** | **"DISABLE"**>	**set_simstate_behavior**	6.53

Table 4—Attribute and command correspondence *(continued)*

HDL attribute name	Attribute value specification	Equivalent UPF command arguments	See
UPF_is_leaf_cell	<**"TRUE"** | **"FALSE"**>	**set_design_attributes -is_leaf_cell**	6.37
UPF_is_macro_cell	<**"TRUE"** | **"FALSE"**>	**set_design_attributes -is_macro_cell**	6.37

The HDL attributes in Table 4 all take values that are string literals. Where a list of names is required, the names in the list should be separated by spaces and without enclosing braces ({}). To attach a UPF attribute to an object in a VHDL context, the UPF attribute shall be declared first, with a data type of `STD.Standard.String` (or the equivalent), before any attribute specification for that attribute.

It shall be an error if any of the attributes in Table 4 is defined multiple times with different values for the same object, regardless of whether the attribute is defined as an HDL attribute or using UPF commands or both.

Examples

A port-supply relationship can be annotated in HDL using the following attributes:

Attribute name: **UPF_related_power_port** and **UPF_related_ground_port**.

Attribute value: "*supply_port_name*", where *supply_port_name* is a string whose value is the simple name of a port on the same interface as the attributed port.

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_related_power_port = "my_VDD",
   UPF_related_ground_port = "my_VSS" *)
output my_Logic_Port;
```

VHDL attribute specification:

```
attribute UPF_related_power_port : STD.Standard.String;
attribute UPF_related_power_port of my_Logic_Port : signal is
"my_VDD";
attribute UPF_related_ground_port : STD.Standard.String;
attribute UPF_related_ground_port of my_Logic_Port : signal is
"my_VSS";
```

Attribute name: **UPF_related_bias_pin**.

Attribute value: "*supply_port_name_list*", where *supply_port_name_list* is a string whose value is a space-separated list of one or more simple names of port(s) on the same interface as the attributed port.

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_related_bias_ports = "my_VNWELL my_VPWELL" *)
output my_Logic_Port;
```

VHDL attribute specification:

```
attribute UPF_related_bias_ports : STD.Standard.String;
attribute UPF_related_bias_ports of my_Logic_Port : signal
is "my_VNWELL my_VPWELL";
```

The same attributes can be specified in UPF, using the **set_port_attributes** command and its generic **-attribute** option, or they can also be specified in UPF using the **set_port_attributes** command and its specific options **-related_power_port**, **-related_ground_port**, and **-related_bias_ports**, respectively (see 6.46).

Isolation clamp value port properties can be annotated in HDL using the following attributes:

Attribute name: **UPF_clamp_value**

Attribute value: <**"0"** | **"1"** | **"Z"** | **"latch"** | **"any"** | **"***value***"**>

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_clamp_value = "1" *) output my_Logic_Port;
```

VHDL attribute specification:

```
attribute UPF_clamp_value : STD.Standard.String;
attribute UPF_clamp_value of my_Logic_Port : signal is "1";
```

The same attributes can be specified in UPF, using the **set_port_attributes** command and its generic **-attribute** option, or it can also be specified in UPF, using the **set_port_attributes** command and its specific option **-clamp_value** (see 6.46).

pg_type port properties can be annotated in HDL using the following attributes:

Attribute name: **UPF_pg_type**

Attribute value: <**"primary_power"** | **"primary_ground"** | **"backup_power"** | **"backup_ground"** >

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_pg_type = "primary_power" *) output myVddPort;
```

VHDL attribute specification:

```
attribute UPF_pg_type : STD.Standard.String;
attribute UPF_pg_type of myVddPort : signal
   is "primary_power";
```

The same attributes can be specified in UPF, using the **set_port_attributes** command and its generic **-attribute** option, or it can also be specified in UPF using the **set_port_attributes** command and its specific option **-pg_type** (see 6.46).

The UPF leaf-cell treatment of a model or instance can be annotated in HDL using the following attributes:

Attribute name: **UPF_is_leaf_cell**

Attribute value: **<"TRUE" | "FALSE"**>

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_is_leaf_cell="TRUE" *) module FIFO (<port list>);
```

VHDL attribute specification:

```
attribute UPF_is_leaf_cell : STD.Standard.String;
attribute UPF_is_leaf_cell of FIFO : entity is "TRUE";
```

The same attribute can be specified in UPF, using the **set_design_attributes** command (see 6.37).

When any register (specified or implied) with the **UPF_retention** attribute value set to **"required"** is included in a power domain that has at least one retention strategy, the register shall be included in a retention strategy defined for the domain.

Elements requiring retention can be attributed in HDL as follows:

Attribute name: **UPF_retention**

Attribute value: <**"required"** | **"optional"**>

SystemVerilog or Verilog-2005 attribute specification:

```
(* UPF_retention = "required" *) module my_mod;
```

VHDL attribute specification:

```
attribute UPF_retention : STD.Standard.String;
attribute UPF_retention of my_flip : variable is "required";
```

The same attribute can be specified in UPF, using the **set_retention_elements** command and its specific option **-retention_purpose** (see 6.51).

5.7 Power state name spaces

Power states are attributed to specific objects in the design. The power states can be referenced by specifying the *object_name*, where *object_name* can be a hierarchical name denoting a power domain, supply set, or supply net. Power states are attributes of the object. Specifically, *power states* of a domain are attributes of the domain and not attributes of the scope of the domain. Thus, an instance may be the scope for multiple domains, each domain containing states with the same name (e.g., `sleep`) without incurring a name space collision.

The following objects may have power states attributed to them:

- Power domains
- Supply sets
- Supply nets
- Supply ports

The **add_power_state** command (see 6.4) is used to define the legal and illegal power states of power domains and supply sets. The `set_power_state` function in the package UPF is used to set the power state of an object during simulation.

The range of possible states for supply nets and ports is defined by the type `supply_net_type` in the package UPF. The state of supply nets and ports can be set through the `assign_supply2supply` or `assign_supply_state` functions in the package UPF. `assign_supply2supply` propagates the association of the source supply net's root supply driver as well as the source's state and voltage values to the destination. `assign_supply_state` is used to assign a supply port that is a root supply driver.

A power state shall be defined before it can be referenced. Semantically, the transition of an object from one power state to another is a power state *event* for the object. The state of a supply net is referenced as a Boolean expression (see 5.4) in the same manner that the state of a logic net is referenced. The power state of a supply set or power domain can be referenced in an expression simply through the supply set or power-domain name.

Examples

```
supply_set_li == SLEEP
-- Returns TRUE if supply_set_li is in a state consistent with state SLEEP

ALU_PD != FULL_OP
-- Returns TRUE if the ALU_PD is in a state inconsistent with FULL_OP
```

5.8 Precedence

To support concise, easily written low-power specifications, UPF commands may range from very specific to very generic in their scope of application. This enables specification of generic defaults that apply widely except where more specific commands provide more focused information. This subclause describes the precedence relations that determine which of several commands that potentially apply in a given situation will actually apply.

A **create_power_domain** command (see 6.17) that explicitly includes a given instance in its extent shall take precedence over one that applies to an instance transitively (i.e., applies to an ancestor of the instance, and therefore to all of its descendants). A **create_power_domain** command that creates an atomic power domain takes precedence over one that creates a non-atomic power domain.

If multiple **set_isolation** commands (see 6.41), or multiple **set_level_shifter** commands (see 6.43), or multiple **set_repeater** commands (see 6.48) potentially apply to the same port, the following criteria (listed in order from highest precedence to lowest precedence) determine the relative precedence of the commands, and only the command(s) with the highest precedence will actually apply:

a) Command that applies to part of a multi-bit port specified explicitly by name

b) Command that applies to a whole port specified explicitly by name

c) Command that applies to all ports of an instance specified explicitly by name

d) Command that applies to all ports of a specified power domain with a given direction

e) Command that applies to all ports of a specified power domain

If multiple strategies of the same type have the same highest precedence, then all of those commands actually apply to the port or part thereof, to the extent allowed by the strategy.

A prefix or suffix to be used to create names for inserted isolation, level-shifter, and repeater cells that is specified by the **–name_prefix** or **–name_suffix** options, respectively, of **set_isolation**, **set_level_shifter**, and **set_repeater**, takes precedence over any user-defined prefix or suffix for these commands specified by the **name_format** command (see 6.35). A prefix or suffix explicitly specified using the **name_format** command in turn takes precedence over the default prefix or suffix specified in the definition of the **name_format** command.

If multiple supply connections potentially apply to the same port, the actual application is determined by the following precedence order, from highest to lowest precedence:

f) Command that explicitly connects to part of a port

g) Command that explicitly connects to a whole port
(e.g., `connect_supply_net -ports/-pins`)

h) Command that automatically connects to ports of an instance (
e.g., `connect_supply_set -connect -elements`)

i) Command that automatically connects to ports of any instance in a given region
(e.g., `connect_supply_set -connect` or
`connect_supply_net -pg_type -domain/-cells`)

Any explicit connection command takes precedence over implicit connections made by default.

For attribute specifications, there is no definition of precedence to select which of several potentially applicable specifications apply. It is an error if any two UPF, HDL, or Liberty attribute specifications provide different values for the same attribute of the same object.

For simstates that apply to a given object at any given time, a more conservative (i.e., more corrupting) simstate takes precedence over a less conservative (less corrupting) simstate.

The following also apply:

j) The precedence of a command is independent of the current scope during the command processing.

k) It shall be an error if the precedence rules fail to uniquely identify the power intent that applies to an object.

l) The **find_objects** command (see 6.26) returns a list of explicit names; these names can refer to whole objects or to elements thereof. When *list* arguments to command options are created using **find_objects**, the level of precedence is based on the expanded value used as the argument, not as the pattern or regular expression used in **find_objects**.

m) The symbol **.** in **–elements {.}** is an explicit reference to the instance corresponding to the current scope.

5.9 Generic UPF command semantics

All **map_*** commands specify the elements to be used in implementation. These specifications override the elements that may be inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, it may not be possible for logical equivalence checking tools to verify the equivalence of the mapped element to its RTL specification.

5.10 effective_element_list semantics

The *effective_element_list* is the set of elements to which a command applies. The *effective_element_list* is constructed from the arguments provided to the command. The terms used in the description of this construction include: *element_list, exclude_list, aggregate_element_list, aggregate_exclude_list, prefilter_element_list,* and *effective_element_list.* The *element_list* and *exclude_list* are lists that contain the elements specified by an instance of the command. The *effective_element_list*, *aggregate_element_list*, and *aggregate_exclude_list* are associated with the named object of the command.

The following arguments can determine the *effective_element_list*:

a) **-elements** *element_list* adds the rooted names in *element_list* to the *aggregate_element_list*. It is not an error for an element to appear more than once in this list.

b) **-model** *model_name* adds the rooted name of each instance that is an instance of the model to the *aggregate_element_list*.

c) **-models** *model_list* or **-model** *model_list* adds the rooted name of each instance that is an instance of any of the models in *model_list* to the *aggregate_element_list*. It is not an error for a model to appear more than once in this list.

d) **-lib** *lib_name* selects all models from the specified *lib_name*. If only **-lib** *lib_name* is specified, the rooted name of each instance that is an instance of every model present in *lib_name* is added to the *aggregate_element_list*.

e) If **-lib** *lib_name* is specified along with **-model** *model_name* or **-models** *model_list*, the model is selected only if it is present in *lib_name*. This results in rooted names for only those models that are present in the *lib_name* library.

f) If **-lib** *lib_name*, **-model**, or **-models** is specified with an **-elements** option, the *aggregate_element_list* is constructed by adding the rooted names from **-elements** and rooted names resulting from any **-lib**/**-model**/**-models** options.

g) **-exclude_elements** *exclude_list* adds the rooted names in *exclude_list* to the *aggregate_exclude_list*. It is not an error for an element to appear more than once in this list. It is not an error for an element in the exclude list to not be in the *aggregate_element_list*.

h) When **-elements** *element_list* is specified with a period (**.**), the current scope is included as a rooted instance in the *aggregate_element_list*.

i) It shall be an error if the *element_list* is not specified as one of **{}**, **{.}**, or **{***list***}**.

j) When **-transitive** is specified with the (default or explicit) value **TRUE**, elements (see 5.10.1) in *aggregate_element_list* that are not leaf cells are processed to include the child elements (see 5.10.2).

k) The *prefilter_element_list* comprises the *aggregate_element_list* with any matching elements from the *aggregate_exclude_list* removed (see 5.10.2).

l) The command arguments identified as filters are predicates that shall be satisfied by elements in the *effective_element_list*. The *prefilter_element_list* is filtered by the predicates to produce the *effective_element_list* (see 5.10.2).

m) The range of legal element types is command dependent for each command that uses **-elements**. Each command specifies the effect of an empty *aggregate_element_list*. An explicitly empty list may be specified with {}.

5.10.1 Transitive TRUE

The detailed semantics of **-transitive TRUE** are described using Figure 3, Figure 4, and Figure 5. The figures are exemplary; the text provides a semantic for the validation of the result.

a) Given a design as shown in Figure 3 with a instance `A` in the current scope, where `A` has child elements `B`, `C`, and `D`; `B` has child elements `E` and `F`, `C` has child elements `G` and `H`, and `D` has child elements `I` and `J`.

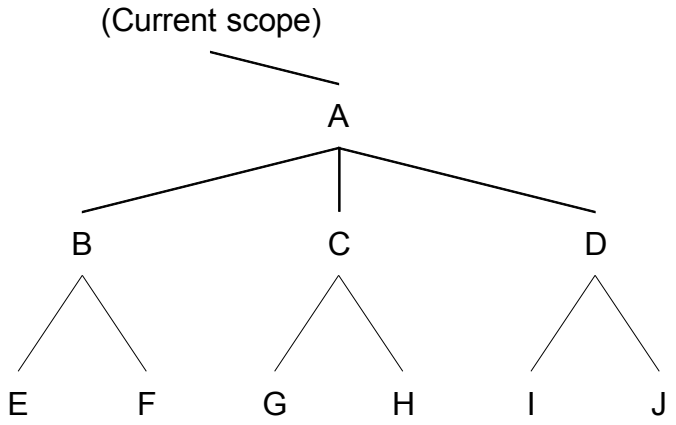

Figure 3—Element processing example design fragment

b) If the specification:

```
-elements {A A/C/H} -exclude_elements {A/C A/D} -transitive TRUE
```

is applied to the design fragment shown in Figure 3, then Figure 4 shows the four specified elements by indicating them as boxed; those specified with exclude are shown with strike-through text.

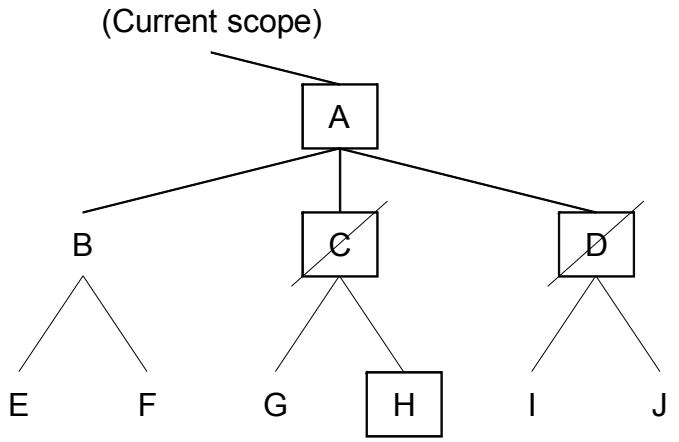

Figure 4—Element processing specification

c) Figure 5 shows the results of the *effective_element_list*. The list includes
`{A A/B A/B/E A/B/F A/C/H}`

The elements included or excluded by transitivity are shown as dashed-boxes or with strike-through text, respectively.

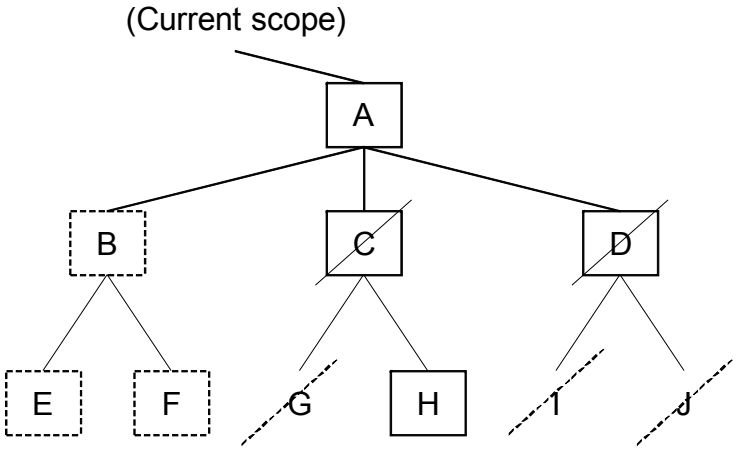

Figure 5—Element processing result

5.10.2 Result

The required result is derived as follows:

```
Begin // at the current scope.
   Initialize by traversing the hierarchy and set element.mark := exclude
   For each element in the aggregate_element_list do
      set element.mark := includeP
      if (transitive = TRUE AND element NOT Leaf_Cell) then
         foreach child in element call mark_child(child, include)
      end if
   done
   For each element in the aggregate_exclude_list do
      set element.mark := excludeP
```

```
        if (transitive = TRUE AND element NOT Leaf_Cell) then
            foreach child in element call mark_child(child, exclude)
        end if
    done
    For each element in the aggregate_element_list call check_and_add(element)
done

proc mark_child(element, value)
    if (element.mark != excludeP AND element.mark != includeP ) then
        element.mark := value
        if (element NOT Leaf_Cell) then
            foreach child in element call mark_child(child, value)
        end if
    end if
end proc

proc check_and_add(element)
    if (element.mark = includeP OR element.mark = include) then
        if (for all filters filter(element) = TRUE) then
            add element to effective_element_list
            if (transitive = TRUE AND element NOT Leaf_Cell) then
                foreach child in element call check_and_add(child)
            end if
        end if
    end if
end proc
```

NOTE—Implementations may use any data structure or algorithm that produces the same results as the preceding method.

5.11 Command refinement

Some UPF commands support incremental refinement. Commands that support incremental refinement are called *refinable commands*. A refinable command may be invoked multiple times on the same object and each invocation may add additional arguments to those specified in previous invocations. The arguments of a refinable command that may be added after the first invocation are called *refining arguments*; these are shown in **boldface-green text** and labeled with an **R** in their respective *arguments* listings. Certain commands have refinable arguments; such arguments may have additional information about that argument added after the first invocation of the command, in much the same way that refinable commands may have additional arguments added later.

The first instance of a refinable command identifies the object to which it applies; all mandatory arguments shall be declared in this call and any other arguments may also be included. Subsequent occurrences of the command that identify the same object shall be executed in the same scope and shall include the **-update** option and refining arguments as required. The mandatory arguments that identify the object to which the command applies (the object name following the command or option name, and for strategies, the domain specification as well) shall also be included in each subsequent occurrence, but other mandatory arguments are not required in subsequent occurrences of the command. The end result will be as if all of the arguments, other than the **-update** argument, had been included in the initial occurrence of the command, either individually (e.g., **-clamp_value** or **-isolation_supply_set**) or merged together into a single argument (e.g., **-elements** or **-exclude_elements**).

For example, the **set_isolation** command (see 6.41) can be invoked for the first time in a given scope to define a strategy name for a particular domain. Subsequent **set_isolation** commands executed in the same scope can specify the same strategy and domain names and also specify additional arguments to further characterize the isolation strategy defined by the previous command. Similarly, the **add_power_state**

command (see 6.4) can be invoked initially in a given scope to define a set of power states for a supply set. A subsequent invocation of **add_power_state** in the same scope and for the same supply set may use the **-update** option to add a **-simstate** specification to each power state definition.

When **–update** is used for command refinement, the following apply:

— It shall be an error if **–update** is specified on the first command of a given kind that applies to a given object.

— It shall be an error if **–update** is not specified on subsequent commands of the same kind that apply to the same object.

— Except for those command arguments that aggregate (see 5.10 and 6.4), it shall be an error if subsequent commands specify a value for a given argument that conflicts with or contradicts a previously specified value for the same argument.

Example

This shows a multiple-part refinement for a usage of **set_isolation** (see 6.41).

a) Constraint specification using port attributes

```
set_port_attributes
    -elements {a b c d}
    -clamp_value 0
```

b) Logical configuration

```
set_isolation demo_strategy -domain pda
    -elements {a b c d}
    -isolation_signal {iso_en}
    -isolation_sense {LOW}
```

c) Adding elements to the strategy

```
set_isolation demo_strategy -domain pda -update
    -elements {e f g}
```

d) Supply set implementation

```
set_isolation demo_strategy -domain pda -update
    -isolation_supply_set pda_isolation_supply
```

The implementation-independent part of the power intent [see item a)] could also be declared in the SystemVerilog HDL using the following attributes:

```
(* UPF_clamp_value = "0" *) out a;
(* UPF_clamp_value = "0" *) out b;
(* UPF_clamp_value = "0" *) out c;
(* UPF_clamp_value = "0" *) out d;
```

In this case, the declaration shall have identical semantics to the equivalent UPF command.

5.12 Error handling

If an error condition occurs, e.g., an incorrect command-line option is specified, then a `TCL_ERROR` exception shall be raised. This exception can be caught using the Tcl `catch` command, so these errors can be prevented from aborting the active **load_upf** command (see 6.28). These errors shall have no impact on further commands. Processing may continue after the error is caught. Sequencing of the error `catch` and the choice of continuation is tool-dependent. The state of the design after an error is not defined. Specifically, a command that raises an error may partially complete before aborting.

In general, all commands that fail shall raise a `TCL_ERROR`. As described in the Tcl documentation, the global variables accessible after an error occurs include *errorCode* and *errorInfo*.

NOTE—The message string returned by the Tcl `catch` command is not specified in this standard.

5.12.1 errorCode

After an error has occurred, this variable contains additional information about the error in a form that is easy to process with programs. *errorCode* consists of a Tcl list with one or more elements. The first element of the list identifies a general class of errors and determines the format of the rest of the list. There are several formats for *errorCode* used by the Tcl code; see also the *Tcl command reference* [B6].

Errors defined in this standard are prefixed with `UPF_`, as shown in the following definitions. Individual applications that implement this standard may define and use additional error codes that do not start with `UPF_`. Implementations need to use errors appropriate to their application.

a) `UPF_RETURN_NOT_VISIBLE error_data`

   This error code indicates the objects referenced in the response of a query are not in the current scope. Queries return object names rooted in the current scope. Because they are called from a current scope that may be different from the scope in which all objects to be returned are visible, it shall be an error if the query cannot represent the objects to be returned as a rooted name.

   The `UPF_RETURN_NOT_VISIBLE` error may be raised in these cases where there are no other errors. When this code is returned, the `error_data` is defined to be the same as the query would have returned, but with fully qualified names for the objects not visible in the current scope.

b) `UPF_QUERY_OBJECT_NOT_DEFINED error_data`

   This error code indicates a query is called with a specific name argument and the named object is not defined in the current scope.

c) `UPF_UPDATE_CONFLICT error_data`

   This error code indicates a command has been called with arguments that conflict with previously specified values.

d) `UPF_UPDATE_MISSING error_data`

   This error code indicates a command has been called without the **-update** argument and the named object has already been defined.

e) `UPF_UPDATE_OBJECT_NOT_FOUND error_data`

   This error code indicates a command has been called with the **-update** argument and the named object has not been previously defined.

f) `UPF_OBJECT_NOT_FOUND error_data`

   This error code indicates a name referenced in a command is not defined in the current scope.

5.12.2 errorInfo

See the *Tcl command reference* [B6].

5.13 Units

Voltage values are expressed as real number literals that represent voltage measurements with the implicit unit of 1 V. For example, the literal 1.3 represents 1.3 V, or equivalently 1300 mV, or 1 300 000 μV.

6. Power intent commands

This clause documents the syntax for each UPF command. For details concerning the simstate semantics, see Clause 9.

6.1 Categories

Each command in this clause is categorized based on the following definitions. Unless otherwise mentioned, all *constructs* (commands and/or options) in this standard are considered *Current*. Constructs considered as *Legacy* or *Deprecated* shall be explicitly denoted.

a) *Current*—A construct defined in the standard with the following characteristics:

   1) It is recommended for use.

   2) Its semantics fully support the latest concepts.

   3) Its interaction with other related constructs is well defined.

   4) It is expected to be part of the standard and be considered for extension in future versions.

b) *Legacy*—A construct defined in the standard with the following characteristics:

   1) It is *not recommended* for use for new code.

   2) Its semantics are not interoperable with all of the latest UPF concepts.

   3) It will not be considered for extensions in future versions.

   4) It is included for backward compatibility only, e.g., **set_isolation -isolation_power_net** (see 6.41).

   Legacy constructs (commands and/or options) have not had their syntax and/or semantics updated to be consistent with other commands in this version of the standard, so their descriptions may contain significant obsolete information and their semantics may not be interoperable with the latest UPF concepts.

c) *Deprecated*—A construct defined in the standard with the following characteristics:

   1) It is *not recommended* for use for any code.

   2) It will not be considered for extensions in future versions.

   3) It may be deleted from future versions, e.g., **merge_power_domains** (see 6.34).

   Deprecated commands are noted in this standard without syntax definitions or semantic explanations. Deprecated options of Current commands are noted in the syntax definition of those commands, but are not mentioned in the semantic explanations of those commands. For more details on any deprecated constructs, see IEEE Std 1801™-2009 [B3].

For recommendations on how to use Current constructs to replace Legacy and Deprecated ones, see Annex D.

6.2 add_domain_elements [deprecated]

This is a deprecated command; see also 6.1 and Annex D.

6.3 add_port_state [legacy]

Purpose	Add states to a port	
Syntax	**add_port_state** *port_name* {**-state** {*name* <*nom* | *min max* | *min nom max* | **off**>}}*	
Arguments	*port_name*	The name of the supply port. Hierarchical names are allowed.
	-state {*name* <*nom* | | *min max*| *min nom max* | **off**>}	The *name* and value for a state of the supply port. The value can be a nominal voltage; a pair specifying the minimum and maximum voltage; a triplet of values specifying the minimum, nominal, and maximum voltages; or **off**.
Return value	Return the fully qualified name (from the current scope) of the created port or raise a `TCL_ERROR` if any of the port states are not added.	

This is a legacy command; see also 6.1 and Annex D.

The **add_port_state** command adds state information to a supply port. If the voltage values are specified, the supply net state is **FULL_ON** and the voltage value is the single nominal value or within the range of min to max; otherwise, if **off** is specified, the voltage value is **OFF**.

It shall be an error if *port_name* does not already exist.

It shall be an error if *nom* < *min* or *max* < *nom*.

Syntax example:

```
add_port_state VN1
   -state {active_state 0.88 0.90 0.92}
   -state {off_state off}
```

6.4 add_power_state

Purpose	Define power state(s) of a power domain or supply set
Syntax	**add_power_state** *object_name* [**-supply** | **-domain**] [**-state** {*state_name* [**-supply_expr** {*boolean_expression*}] [**-logic_expr** {*boolean_expression*}] [**-simstate** *simstate*] [**-legal** | **-illegal**]}]* [**-complete**] [**-update**]

Arguments	*object_name*	Simple name of a power domain or supply set.	
	-supply | **-domain**	These arguments specify the kind of object to which this command applies. If **-supply** is specified, the *object_name* shall be the name of a supply set or a supply set handle. If **-domain** is specified, the *object_name* shall be the name of a power domain. If neither is specified, the type of *object_name* determines the kind of object to which the command applies.	
	-state {*state_name ...*}	*state_name* is the simple name of the state being defined or refined.	
	-supply_expr {*boolean_expression*}	**-supply_expr** specifies a Boolean expression defined in terms of supply ports, supply nets, and/or supply set handle functions that evaluates to *True* when the object is in the state being defined.	**R**
	-logic_expr {*boolean_expression*}	**-logic_expr** specifies a Boolean expression defined in terms of logic nets and/or power states of supply sets and/or power domains that evaluates to *True* when the object is in the state being defined.	**R**
	-simstate *simstate*	**-simstate** specifies a **simstate** for the power states associated with a supply set. Valid values are **NORMAL**, **CORRUPT_ON_CHANGE**, **CORRUPT_STATE_ON_CHANGE**, **CORRUPT_STATE_ON_ACTIVITY**, **CORRUPT_ON_ACTIVITY**, **CORRUPT**, and **NOT_NORMAL**. See 4.4.2.6.	**R**
	-legal | **-illegal**	These options specify the legality of the state being defined as either legal or illegal. The default is **-legal**.	**R**
	-complete	Specifies that all power states to be defined for this object have been defined. This implies that all legal power states have been defined and any state of the object that does not match a defined state is an illegal state.	**R**
	-update	Indicates this command provides additional information for a previous command with the same *object_name* and executed in the same scope.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

Semantics

add_power_state defines one or more power states of an object. Each power state definition is independent of any other power state definition. Two different power states of the same object may have intersecting or overlapping **-supply_expr** and/or **-logic_expr** expressions. Such states may have different legalities. A power domain or a supply set may be in a state that matches more than one power state definition.

Multiple power states can be defined for an object in a single call to this command.

The power states defined for a given object include only those defined explicitly for that object [and for power states of a supply set or supply set handle, the default power states **DEFAULT_NORMAL** and **DEFAULT_CORRUPT** (see 4.6.3)]. Power states defined for one object are not inherited implicitly by any related object (e.g., by a supply set handle with which a supply set has been associated or vice versa). However, power states of one object may be defined in terms of power states of another object, to represent dependencies or correlation of power states.

The set of power states for a given object may be specified incrementally by using **-update**. The first **add_power_state** for that object may define one or more power states. Subsequent **add_power_state** **-update** commands for the same object may define additional power states.

A power state definition itself may also be specified incrementally by using **-update**. The initial definition of the power state defines at least the power state name and may specify additional information about this power state. Subsequent **add_power_state** **-update** commands for the same power state of the same object may specify additional details about that power state.

If a power state definition defined with a **-supply_expr** is updated with another **-supply_expr**, the definition becomes the conjunction of the two.

supply_expr' = (*previous supply_expr*) && (**-update** *supply_expr*)

Similarly, if a power state definition defined with a **-logic_expr** is updated with another **-logic_expr**, the definition becomes the conjunction of the two.

logic_expr' = (*previous logic_expr*) && (**-update** *logic_expr*)

A logical contradiction exists when a logic net or supply set or power domain is specified to be more than one value for the state, e.g., `(enable == '1')` and `(enable == '0')`. A power state definition is *erroneous* if it contains logical contradiction(s).

The **-logic_expr** *boolean_expression* shall be a Boolean expression (see 5.4) referencing control signals, clock signal intervals, and/or power states of a supply set or power domain. For convenience, the following expression forms may appear in this expression:

a) ***interval(****signal_name edge1 edge2****)***

Equivalent to
the time between the most recent two specified edges of *signal_name*
(returns the largest supported time value until both edges have occurred)

where
edge1, *edge2* shall be one of **posedge** or **negedge**.

b) ***interval(****signal_name edge****)***

Equivalent to
interval(*signal_name edge edge****)***

c) ***interval(****signal_name****)***

Equivalent to
interval(*signal_name* **posedge posedge*)***

d) *supply_set* == *power_state*

Equivalent to
{ logic_expression && supply_expression }

where
logic_expression and *supply_expression* are the *boolean_expressions* used to define the *power_state* of *supply_set*.

e) *power_domain* == *power_state*

Equivalent to
{ logic_expression }

where
logic_expression is the *boolean_expression* used to define the *power_state* of *power_domain*.

Examples

```
-logic_expr { enable == 1'b1 && interval(clk) < 5ps }
-logic_expr { core_pd.primary == ON_1d2v }
-logic_expr { core_pd == turbo && ram_pd != sleep }
```

Within a logic expression specified as part of a power state definition for a given power domain, the supply set handles of that power domain may be referenced directly without prefixing the name with the supply set or supply set handle name. To refer to an object declared in the current scope with the same name as a supply set handle of the power domain, the object name shall be prefixed with `./`.

The **-supply_expr** *boolean_expression* shall be a Boolean expression (see 5.4) that may reference supply nets, supply ports, and/or functions of supply sets or supply set handles. For convenience, the following expression forms may appear in this expression:

f) *supply_net* == *net_state*

Equivalent to
{ supply_net.state == net_state }

where
supply_net is the name of a supply port or net or a supply set (handle) function
net_state is the name of a state associated with *supply_net*.

g) *supply_net* == *{ net_state min_voltage max_voltage }*

Equivalent to
{ supply_net.state == net_state &&
min_voltage <= supply_net.voltage && supply_net.voltage <= max_voltage }

where
supply_net is the name of a supply port or net or a supply set (handle) function (see 4.6.1)
net_state is the name of a state associated with *supply_net*.

h) *supply_net* == *{ net_state nom_voltage }*

Equivalent to
{ supply_net == { net_state min_voltage max_voltage }} for verification.

where
supply_net is the name of a supply port or net or a supply set (handle) function
net_state is the name of a state associated with *supply_net*
min_voltage = *nom_voltage* – *threshold*
max_voltage = *nom_voltage* + *threshold*
threshold = 0.000001 * 10^(6 – *min*(6,*sigdigits*)) / 2
sigdigits = # of significant digits to the right of the decimal point in *nom_voltage*.

This form is for verification only; it is an error if it is used for implementation.

It is an error if *min_voltage* > *max_voltage*.

NOTE 1—The value of the Tcl variable `tcl_precision`, which specifies how many digits of precision are preserved when converting a floating-point number to a string, may affect the result of the preceding transformation if it is set to a number less than *sigdigits*.

i) *supply_net* == *{ net_state min_voltage nom_voltage max_voltage }*

Equivalent to
{ supply_net == { net_state min_voltage max_voltage }} for verification.
Implementation tools may use all three values to help make implementation choices.

It is an error if *min_voltage* > *nom_voltage* or *nom_voltage* > *max_voltage*.

Examples

`{VDD == { FULL_ON 0.8 } }` is equivalent to `{ VDD == { FULL_ON 0.75 0.85 }}`
`{Pwr == { FULL_ON 0.925 } }` is equivalent to `{ Pwr == { FULL_ON 0.9245 0.9255 }}`
`{Gnd == { FULL_ON 0.00 } }` is equivalent to `{ Gnd == { FULL_ON -0.005  0.005 }}`

Within a supply expression specified as part of a power state definition for a given supply set or supply set handle, the functions of that supply set or supply set handle may be referenced directly without prefixing the name with the power domain name. To refer to an object declared in the current scope with the same name as a function of the supply set or supply set handle, the object name shall be prefixed with `./`.

Restrictions

j) If a supply expression is used to define a power state of a given supply set or supply set handle, it shall only refer to supply ports, supply nets, and/or functions of the given supply set or supply set handle. It is an error if such a supply expression refers to functions of another supply set or supply set handle.

k) If a logic expression is used to define a power state of a given supply set or supply set handle, it shall only refer to logic ports, logic nets, interval functions, and/or power states of the given supply set or supply set handle. It is an error if such a logic expression refers to functions of a supply set or supply set handle, power states of another supply set or supply set handle, or power states of a domain.

l) If a logic expression is used to define a power state of a given power domain, it shall only refer to logic ports, logic nets, interval functions, power states of supply sets or supply set handles, and/or power states of other power domains. It is an error if such a logic expression refers to supply ports, supply nets, or functions of a supply set or supply set handle.

m) It is an error if a supply expression is used to define a power state of a power domain.

n) It is an error if a simstate is associated with a power state of a power domain.

o) When **-simstate**

  1) is first specified for a named state, any of the arguments may appear.

  2) is specified as **NOT_NORMAL**, the effect shall be the same as if **CORRUPT** had been specified, see (4.6.3), except that the definition may be subsequently refined to any simstate other than **NORMAL**.

  3) has previously been specified as **NORMAL**, **CORRUPT**, **CORRUPT_ON_ACTIVITY**, **CORRUPT_ON_CHANGE**, **CORRUPT_STATE_ON_CHANGE**, or **CORRUPT_STATE_ON_ACTIVITY**, it shall be an error if an **add_power_state -update** command for the same object specifies any simstate other than that originally specified (e.g., once **CORRUPT** has been specified for a particular state, it shall remain as **CORRUPT** in any subsequent updates for the definition of that state).

p) The simstate for **DEFAULT_NORMAL** is **NORMAL**.

q) The simstate for **DEFAULT_CORRUPT** is **CORRUPT**.

r) There is no default simstate for a user-defined power state.

s) The supply set is in the **DEFAULT_CORRUPT** power state when it is not in one of the defined power states of the supply set that have simstates defined on them, including the **DEFAULT_NORMAL** predefined state.

t) If **-illegal** has been specified in the definition of a power state for a given object, it is an error if that object is in a state that matches the definition of that power state. A verification tool shall emit an error message when an object is in an illegal power state.

u) If **-complete** has been specified in an **add_power_state** command for a given object, it is an error if that object is in a state that does not match any of the defined power states. A verification tool shall emit an error message when an object is in such an undefined state.

v) If **-complete** has been specified on an **add_power_state** command for a given object, it is an error if a subsequent update to that command is executed.

NOTE 2—The choice of state name has no simstate implications.

NOTE 3—Implementation tools may optimize a design based on the presumption illegal states never occur. Such optimizations are allowed only if they do not change the behavior of the design.

NOTE 4—If the **add_power_state** command for the primary supplies of two interconnected domains are both defined as complete, this implies that all intended legal states have been defined for each domain, and, therefore, all possible state combinations of the two domains have been defined.

Syntax examples:

```
add_power_state PdA.primary -supply
-state {GO_MODE
    -logic_expr SW_ON -simstate NORMAL
    -supply_expr {(power == {FULL_ON 0.8})
    && (ground == {FULL_ON 0})
    && (nwell == {FULL_ON 0.8})}
-state {OFF_MODE
    -logic_expr {!SW_ON}
    -supply_expr {power == {OFF}}
    -simstate CORRUPT}
-state {SLEEP_MODE
    -logic_expr {SW_ON && (interval(clk_dyn posedge negedge) >= 100ns)}
    -supply_expr {(power == {FULL_ON 0.8})
    && (ground == {FULL_ON 0})
    && (nwell == {FULL_ON 1.0})}
    -simstate CORRUPT_STATE_ON_CHANGE}

add_power_state PdA.primary -supply -update -complete

add_power_state PdTOP -domain
    -state {GOGO -logic_expr {u1/PdA.primary == GO_MODE}}
add_power_state PdTOP -state {GOGO -legal} -update
```

6.5 add_pst_state [legacy]

Purpose	Define the states of each of the supply nets for one possible state of the design	
Syntax	**add_pst_state** *state_name* **-pst** *table_name* **-state** *supply_states*	
Arguments	*state_name*	The simple name of the state being defined.
	-pst *pst_name*	The power state table (PST) to which this state applies.
	-state *supply_states*	The list of supply net state names (see 6.20), listed in the corresponding order of the **-supplies** listing in the **create_pst** command (see 6.19). A ***** in place of a state name indicates this is a “don’t care” for that supply.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

This is a legacy command; see also 6.1 and Annex D.

The **add_pst_state** command defines the name for a specific state of the supply nets defined for the PST *table_name*.

It shall be an error if

— The number of *supply_states* is different from the number of supply nets within the PST.

— A *state_name* is defined more than once for the same PST.

Syntax example:

```
create_pst            pt -supplies { PN1   PN2   SOC/OTC/PN3 }
add_pst_state s1 -pst pt -state    { s08   s08   s08         }
add_pst_state s2 -pst pt -state    { s08   s08   off         }
add_pst_state s3 -pst pt -state    { s08   s09   off         }
```

6.6 apply_power_model

Purpose	Connects the interface supply set handles of a previously loaded power model	
Syntax	**apply_power_model** *power_model_name* [**-elements** *instance_list*] [**-supply_map** {{*lower_scope_handle upper_scope_supply_set*}*}]	
Arguments	*power_model_name*	The name a previously defined power model. See 6.8.
	-elements *instance_list*	The list of instances to which the specified power model applies.
	-supply_map {{*lower_scope_handle upper_scope_supply_set*}*}	How the interface supply handles of the corresponding power model connect with the actual supply sets or supply set handles in the current scope.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **apply_power_model** command describes the connections of the interface supply set handles of a previously loaded power model with the supply sets in the scope where the corresponding macro cells are instantiated.

If **-elements** is not specified, the specified supply association is applied to all instantiations of targeted macro cells by the specified power model (see 6.8) under the current scope. The general precedence rules in 5.8 apply here as well.

Each pair in the **-supply_map** option implies an **associate_supply_set** command (see 6.7) of the following general form:

```
associate_supply_set upper_scope_supply_set
-handle lower_scope_handle
```

The arguments of the **-supply_map** option need to be such that the implied **associate_supply_set** commands are legal.

The supply connection specified by **-supply_map** overwrites any implicit or automatic supply set connection. It is an error if a specified supply connection in **-supply_map** conflicts with any explicit connections.

The following also apply:

a) It is an error to update any power intent command specified within a power model from outside of this model.

   If a power model has any power state specified with simstate **NOT_NORMAL**, it cannot be updated with a specific simstate from commands outside of the power model. As a result, the **CORRUPT** simulation semantics shall apply to the power state (see 9.4).

b) The processing of this command shall follow the description in Clause 8.

c) When **apply_power_model** is used with **-elements**, it is an error if the underlying cell name of each instance does not match the corresponding macro cell name specified in the **-for** option of **begin_power_model** (see 6.8) or the *power_model_name* when the **-for** option (of **begin_power_model**) is not specified.

Syntax example:

```
apply_power_model upf_model -elements I1 -supply_map {{PD.ssh1 ss1} {PD.ssh2
   ss2}}
```

For other examples of using these commands, see Annex E.

6.7 associate_supply_set

Purpose	Associate a supply set with a power domain, power switch, or strategy supply set handle	
Syntax	**associate_supply_set** *supply_set_name* **-handle** *supply_set_handle*	
Arguments	*supply_set_name*	The rooted name of a supply set to associate with a supply set handle.
	-handle *supply_set_handle*	The supply set handle with which the supply set is associated.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **associate_supply_set** command associates a supply set with a power domain, power switch, or strategy supply set handle (see 5.3.3.2). As a result, each function of the named supply set is associated with the corresponding function of the supply set handle, which makes the named supply set and the supply set handle equivalent (see 4.4.3). Both the *supply_set_name* and the *supply_set_handle* shall refer to predefined or previously created supply sets.

The *supply_set_handle* may be a predefined supply set handle. The predefined supply set handles are as follows:

a) The predefined supply set handles for a power domain *domain_name* (see 6.17) include: *domain_name*.**primary**, *domain_name*.**default_retention**, and *domain_name*.**default_isolation**. User-defined names for *supply_set_handle* are also permitted.

b) The predefined supply set handle for a power-switch *switch_name* (see 6.18) is *switch_name*.**supply**.

c) The predefined supply set handles for an isolation cell strategy *isolation_name* (see 6.41) are *domain_name*.*isolation_name*.**isolation_supply_set**, if there is only one isolation supply set, or *domain_name*.*isolation_name*.**isolation_supply_set[***index***]**, where *index starts* at `0`, if there are multiple isolation supply sets. The named supply set may be associated with one of these supply set handles using the **associate_supply_set** command as follows:

```
associate_supply_set U1/PD1.my_iso.isolation_supply_set\[1\]
   -handle U1/PD1.my_iso.clamp
```

d) The predefined supply set handles for a level-shifter strategy *level_shifter_name* (see 6.43) are *domain_name.level_shifter_name*.**input** and *domain_name.level_shifter_name*.**output**.

e) The predefined supply set handle for a retention strategy *retention_name* (see 6.49) is *domain_name.retention_name*.**supply**.

It shall be an error if

— The supply set handle is defined for a strategy and more than one supply set is associated with that supply set handle.

— The supply set handle is defined for a power domain, and more than one supply set defined in an ancestor scope is associated with that supply set handle, or more than one supply set defined in the scope of the power domain or a descendant scope is associated with that supply set handle.

— The associations of supply sets with supply set handles form a loop of associations.

Syntax examples:

```
associate_supply_set some_supply_set
   -handle U1/PD1.mem_ss
```

NOTE—A supply set handle can also appear as the *supply_set_name* in an **associate_supply_set** command. This allows transitive association of supply sets, such as the following:

```
associate_supply_set top_level_SS -handle PD1.primary

associate_supply_set PD1.primary -handle PD2.backup

associate_supply_set PD1.primary -handle PD3.default_isolation
```

6.8 begin_power_model

Purpose	Define a power model	
Syntax	**begin_power_model** *power_model_name* [**-for** *model_list*]	
Arguments	*power_model_name*	The name of the power model.
	-for *model_list*	The names of the hard IP or macro cells to which the power model applies.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

The **begin_power_model** and **end_power_model** (see 6.25) commands define a power model containing other UPF commands. A power model is used to define the power intent of a hard IP and shall be used in conjunction with one or more model representations. A power model defined with **begin_power_model** is terminated by the first subsequent occurrence of **end_power_model** in the same UPF file.

-for indicates that the power model represents the power intent for a family of model definitions. When **-for** is not specified, the *power_model_name* shall also be a valid macro cell name. It is an error if the targeted model has a `UPF_is_leaf_cell` attribute set to `FALSE`. It is also an error if any design objects referred to in a power model cannot be found in the corresponding library model or behavioral model of the cell.

A power model can be referenced by its simple name from anywhere in a power intent description. It is an error to have two power models with the same name.

It is also an error if the following commands are specified within the model:

- **name_format** (see 6.35)
- **save_upf** (see 6.36)
- **set_scope** (see 6.52)
- **load_upf -scope** (see 6.28)
- **begin_power_model** (see 6.8)
- **apply_power_model** (see 6.6)
- Any deprecated/legacy commands/options (see Annex D)

To specify supplies coming into or out of the model, or a supply that has at least one data port related to it, use the **-supply** option of the **create_power_domain** command (see 6.17) for the top-scope power domain of the power model. In addition, the system power states defined upon these supply set handles become the power state definition at the interface of the power model, which shall be consistent with the upper-scope system power states into which the corresponding upper-scope supply sets are mapped (see 6.6). The defined supply set handles are also called *interface supply handles* of the power model. Finally, the simstate simulation semantics described in 9.4 applies to all supply sets or supply set handles defined within a power model.

All power commands within a power model describe power intent that has already been implemented with the targeted cells. No new logic or design objects shall be inferred within the cell instances targeted by a power model.

A power model can be applied to specific instances using **apply_power_model** (see 6.6). A power model that is not referenced by an **apply_power_model** command does not have any impact on the power intent of the design.

Syntax example:

```
begin_power_model upf_model -for cellA
   create_power_domain PD1 -elements {.} -supply {ssh1} -supply {ssh2}
   ;# other commands …
end_power_model
```

For more examples of using these commands, see Annex E.

6.9 bind_checker

Purpose	Insert checker modules and bind them to instances
Syntax	**bind_checker** *instance_name* **-module** *checker_name* [**-elements** *element_list*] [**-bind_to** *module* [**-arch** *name*]] [**-ports** {{*port_name net_name*}*}]

Arguments	*instance_name*	The name used to instance the checker module in each instance.
	-module *checker_name*	The name of a SystemVerilog module containing the verification code. The verification code itself shall be written in SystemVerilog, but it can be bound to either a SystemVerilog or VHDL instance.
	-elements *element_list*	The list of instances.
	-bind_to *module* [**-arch** *name*]	The SystemVerilog module or VHDL entity/architecture for which all instances are the target of this command.
	-ports {{*port_name net_name***}*}**	The association of signals to the checker's ports.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **bind_checker** command inserts checker modules into a design without modifying the design code or introducing functional changes. The mechanism for binding the checkers to instances relies on the SystemVerilog `bind` directive. The `bind` directive causes one module to be instantiated within another, without having to explicitly alter the code of either. This facilitates the complete separation between the design implementation and any associated verification code.

Signals in the target instance are bound by position to inputs in the bound checker module through the port list. Thus, the bound module has access to any and all signals in the scope of the target instance, by simply adding them to the port list, which facilitates sampling of arbitrary design signals.

If **-bind_to** is specified, an instance of checker is created in every instance of the module. Otherwise, an instance of the checker is only created within the current scope.

port_name is a port defined on the interface of *checker_name* and *net_name* is a name of a net relative to the scope where *checker_name* is being instantiated.

It shall be an error if *instance_name* already exists in **-bind_to** *module*.

This command is for verification only; implementation tools shall ignore it.

Syntax example:

```
bind_checker chk_p_clks
-module assert_partial_clk
-bind_to aars
-ports {{prt1 clknet2} {port3 net4}}
```

Modeling mutex assertions

To model mutex assertions (see 6.50 and 6.49), the assertions can be put in a SystemVerilog `checker_module` with following interface:

```
module checker_module ( save, restore, reset_a, clock_a );
input save, restore, reset_a, clock_a;
... different mutex assertions ...
endmodule
```

The **bind_checker** command would look like the following:

```
bind_checker mutex_checker_inst -module checker_module \
-ports { {save PDA.test_retention.save_signal } \
{ restore PDA.test_retention.restore_signal } \
{ reset_a reset_a } \
{ clock_a clock_a } }
```

6.10 connect_logic_net

Purpose	Connect a logic net to logic ports	
Syntax	**connect_logic_net** *net_name* **-ports** *port_list* [**-reconnect**]	
Arguments	*net_name*	A simple name.
	-ports *port_list*	A list of ports on the interface of the current scope and/or on instances that are located in the current scope and its descendants.
	-reconnect	Allows a port that is already connected to a net to be disconnected from the existing net and connected to *net_name*.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **connect_logic_net** command connects a logic net to the specified ports. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant subtree of the active UPF scope as required to support connections created by **connect_logic_net** (see 9.2). The connection from *net_name* in the active UPF scope to any element in *port_list* shall not cross any power-domain boundaries.

The net and ports shall be of a compatible type. The following HDL types are compatible with each other:

— SystemVerilog `logic`
— VHDL `std_ulogic`

It shall be an error if:

a) *net_name* is not the name of a logic net defined in the current HDL scope either explicitly or implicitly as a result of a **create_logic_net** command.
b) A HighConn port in *port_list* is already connected to a different net than *net_name*, unless the **-reconnect** option is specified.
c) A LowConn port in *port_list* is already connected to a different net than *net_name*.
d) The same port name occurs in the *port_list* of multiple **connect_logic_net** commands with different *net_name* arguments.

NOTE 1—Use **create_logic_port** (see 6.16) to create new logic ports on power-domain boundaries.

NOTE 2—This command exists to allow for the propagation of signals from a power-management block. Using this command to provide non-power control connections may cause the logic function to diverge from the HDL and is strongly discouraged.

Syntax example:

```
connect_logic_net tree_top
  -ports {s b}
```

6.11 connect_supply_net

Purpose	Connect a supply net to supply ports	
Syntax	**connect_supply_net** *net_name* [**-ports** *ports_list*] [**-pg_type** {*pg_type_list element_list*}]* [**-vct** *vct_name*] [**-cells** *cells_list*] [**-domain** *domain_name*] [**-pins** *pins_list*] [**-rail_connection** *rail_type*]	
Arguments	*net_name*	A simple name.
	-ports *ports_list*	A list of rooted port names.
	-pg_type {*pg_type_list element_list*}	An indirect connection specification via the *pg_type* on the instance's ports.
	-vct *vct_name*	A value conversion table (VCT) defining how values are mapped from UPF to an HDL model or from the HDL model to UPF.
	-cells *cells_list*	A list of cells to use for **-pg_type** or **-rail_connection**.
	-domain *domain_name*	The domain indicates the scope to use for **-pg_type** or **-rail_connection**.
Deprecated arguments	**-pins** *pins_list*	A list of pins on cells to connect. This is a deprecated option; see also 6.1 and Annex D.
	-rail_connection *rail_type*	The rail type (for older libraries). This is a deprecated option; see also 6.1 and Annex D.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **connect_supply_net** command connects a supply net to the specified ports. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant subtree of the current scope as required to support supply port/net connections made explicitly, automatically, or implicitly (see 9.2) This explicit connection overrides (has higher precedence than) the implicit and automatic connection semantics (see 9.2) that might otherwise apply. **-domain** or **-cells** is required when the **-pg_type** option is specified.

If **connect_supply_net** is used to connect a supply net defined with `create_supply_net -domain D` (see 6.20) to a pg pin of an instance, then the instance shall be in the extent of power domain `D`.

Use the following:

-ports to connect to supply ports.

-cells to connect to all pins of the appropriate type (power or ground) on the specified cells.

-pg_type to connect to ports on the instances that have the specified *pg_type*.

-vct to indicate that for every HDL port to which the net is connected, the supply net state shall be converted if it is being propagated into the HDL port (see 6.23) or the HDL port value shall be converted if it is being propagated onto the supply net (6.14). **-vct** is ignored for any connections of the supply net to supply ports defined in UPF.

The following also apply:

- It shall be an error if any cell, domain, port, supply net, or instance specified in this command does not exist.
- It shall be an error if the value conversions specified in the value conversion table (VCT) do not match the type of the HDL port.
- It shall be an error if neither **-ports** nor **-pg_type** is specified in a **connect_supply_net** command.
- The **-ports** option is mutually exclusive with the **-cells**, **-domain**, and **-pg_type** options.
- Automatic propagation of a supply net throughout the extent of a power domain is determined by its usage within the domain, such as primary supply, retention supply, etc.
- It shall be an error if *net_name* has not been previously created; in this case, a `0` shall be returned.
- If **-pg_type** is specified, it shall be an error if an instance does not exist or the specified attribute does not exist on any port of the instance.

Syntax examples:

```
connect_supply_net fb
  -ports {jk jb}

connect_supply_net mc
  -ports {rl}
  -vct SV_TIED_HI
```

The following command connects the supply net `VDDX` to the `VDD` port of a hierarchical instance `I1/I2`:

```
connect_supply_net VDDX -ports I1/I2/VDD
```

The following command connects the supply net `VDDX` to the `VDD` ports of all instances within hierarchical instance `I1/I2`:

```
connect_supply_net VDDX -ports [find_objects I1/I2 -pattern "*/VDD" -
   object_type port]
```

6.12 connect_supply_set

Purpose	Connect a supply set to particular elements	
Syntax	**connect_supply_set** *supply_set_ref* {**-connect** {*supply_function pg_type_list*}}* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-transitive** [<**TRUE** | **FALSE**>]]	
Arguments	*supply_set_ref*	The rooted name of the supply set.
	-connect {*supply_function pg_type_list*}	Define automatic connectivity for a *supply_function* of the *supply_set_ref* as ports having the specified *pg_type_list* attributes (see 6.11).
	-elements *element_list*	The list of instance names to add.
	-exclude_elements *exclude_list*	The list of instances to exclude from the *effective_element_list*.

Arguments	**-transitive** [<**TRUE** | **FALSE**>]	If **-transitive** is not specified at all, the default is **-transitive TRUE**. If **-transitive** is specified without a value, the default value is **TRUE**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **connect_supply_set** command connects a supply set to the specified elements. The nets of the set are propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant subtree of the current scope as required to implement the supply net connection (see 9.2) This explicit connection overrides (has higher precedence than) the implicit and automatic connection semantics (see 9.2) that might otherwise apply.

This command applies to elements in the *effective_element_list* (see 5.10) as follows:

a) When *supply_set_ref* refers to a handle associated with a domain, the *prefilter_element_list* is filtered to only include elements within the extent of the domain.

b) When *supply_set_ref* refers to a handle associated with a strategy, the *prefilter_element_list* is filtered to only include all elements connected to the strategy's supply.

c) When *supply_set_ref* refers to a handle associated with a domain and the *aggregate_element_list* is empty, all elements in the extent of the domain are added to the *aggregate_element_list*.

d) When *supply_set_ref* refers to a handle associated with a strategy and the *aggregate_element_list* is empty, all elements connected to the respective strategy supply are added to the *aggregate_element_list*.

-connect is additive, i.e., on a particular supply function, a subsequent invocation setting *pg_type_list* adds the additional *pg_type_list*.

NOTE—The *exclude_list* in **-exclude_elements** can specify elements that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

It shall be an error if

— A particular *pg_type_list* is associated with more than one supply net for any given instance in **-connect**.

— More than one supply net is connected to the same port in an instance, even if the connection is the result of more than one command that connects supply nets, e.g, **connect_supply_set**, **connect_supply_net**, etc.

— Any element of *element_list* or *exclude_list* is not in a specified domain or strategy referenced in the *supply_set_handle*.

Syntax example:

```
connect_supply_set some_supply_set
   -elements {U1/U_mem}
   -connect {power {primary_power}}
   -connect {ground {primary_ground}}
```

6.13 create_composite_domain

Purpose	Define a composite domain comprised of one or more subdomains		
Syntax	**create_composite_domain** *composite_domain_name* [**-subdomains** *subdomain_list*] [**-supply** {*supply_set_handle* [*supply_set_ref*]}] [**-update**]		
Arguments	*composite_domain_name*	The name of the composite domain; this shall be a simple name.	
	-subdomains *subdomain_list*	The **-subdomains** option specifies a list of rooted domain names, including any previously created composite domains.	**R**
	-supply {*supply_set_handle* [*supply_set_ref*]}	The **-supply** option specifies the *supply_set_handle* for *composite_domain_name*. If *supply_set_ref* is also specified, the domain *supply_set_handle* is associated with the specified *supply_set_ref*. The *supply_set_ref* may be any supply set visible in the current scope. See also 6.7.	**R**
	-update	Use **-update** if the *composite_domain_name* has already been defined.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

A *composite power domain* is a simple container for a set of power domains. Unlike a power domain, a composite domain has no corresponding physical region on the silicon. Attributes like power states and the primary *supply_set_handle* can be specified on a composite domain, but these attributes shall not be applied to subdomains. However, operations performed on the composite domain shall be applied to each subdomain, e.g., defining a strategy.

The following commands, applied to a composite domain, are applied to each subdomain if and only if the application of that command does not result in an error in any subdomain:

connect_supply_net

map_power_switch

map_retention_cell

set_isolation

set_level_shifter

set_repeater

set_retention

use_interface_cell

Only the primary supply handle can be specified in the **-supply** option. The following also apply:

a) Composite power domains can be used as a subdomain of other composite power domains.

b) Since a composite domain is simply a container, commands can still be applied to subdomains after composition.

c) For each subdomain: If a supply set is associated with the primary *supply_set_handle* of a subdomain, that supply set shall be equivalent to the primary supply set of the composite domain or declared as equivalent to the primary supply set of the composite domain (see also 6.40).

d) Commands applied to a subdomain do not affect any other subdomain or the composite domain.

e) Subdomains of a composite domain can still be referenced after composition, in the sense their elements lists are valid after composition, and all aspects of the subdomain (e.g., strategies defined on them) can be referenced.

When the primary *supply_set_handle* and a *supply_set_ref* are specified in **-supply**, it is equivalent to the following:

```
associate_supply_set supply_set_ref
   -handle composite_domain_name.primary
```

Syntax example:

```
create_composite_domain my_combo_domain_name
   -subdomains {a/pd1 b/pd2}
   -supply {primary could_be_on_ss}
```

6.14 create_hdl2upf_vct

Purpose	Define a VCT that can be used in converting HDL logic values into `state` type values	
Syntax	**create_hdl2upf_vct** *vct_name* **-hdl_type** {<**vhdl** | **sv**> [*typename*]} **-table** {{*from_value to_value*}*}	
Arguments	*vct_name*	The VCT name.
	-hdl_type {<**vhdl** | **sv**> [*typename*]}	The HDL type for which the value conversions are defined.
	-table {{*from_value to_value*}*}	A list of the values of the HDL type to map to UPF `state` type values.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_hdl2upf_vct** command defines a value conversion table (VCT) from an HDL logic type to the `state` type of the supply net value (see Annex B) when that value is propagated from HDL port to a UPF supply net. It shall provide a conversion for each possible logic value that the HDL port can have. **create_upf2hdl_vct** does not check that the set of HDL values are complete or compatible with any HDL port type.

vct_name provides a name for the value conversion table for later use with the **connect_supply_net** command (see 6.11). A VCT can be referenced by its simple name from anywhere in a power intent description. It is an error to have two VCTs with the same name.The predefined VCTs are shown in Annex F.

-hdl_type specifies the HDL type for which the value conversions are defined. This information allows a tool to provide completeness and compatibility checks. If the *typename* is not one of the language's predefined types or one of the types specified in the next paragraph, then it shall be of the form *library*.*pkg*.*type*.

The following HDL types shall be the minimum set of types supported. An implementation tool may support additional HDL types.

a) VHDL
   1) `Bit`, `std_[u]logic`, `Boolean`
   2) Subtypes of `std_[u]logic`
b) SystemVerilog
   `reg/wire`, `Bit`, `Logic`

-table defines the 1:1 conversion from HDL logic value to the UPF partially on and on/off states. The values are consistent with the HDL type values.

For example:

— When converting from SystemVerilog *logic type*, the legal values are `0`, `1`, `X`, and `Z`.
— When converting from SystemVerilog or VHDL `bit`, the legal values are `0` or `1`.
— When converting from VHDL `std_[u]logic`, the legal values are `U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, and `-`.

The conversion values have no semantic meaning in UPF. The meaning of the conversion value is relevant to the HDL model to which the supply net is connected.

Syntax examples:

```
create_hdl2upf_vct
   vlog2upf_vss
   -hdl_type {sv reg}
   -table {{X OFF} {0 FULL_ON} {1 OFF} {Z PARTIAL_ON}}
create_hdl2upf_vct
   stdlogic2upf_vss
   -hdl_type {vhdl std_logic}
   -table {{'U' OFF}
        {'X' OFF}
        {'0' OFF}
        {'1' FULL_ON}
        {'Z' PARTIAL_ON}
        {'W' OFF}
        {'L' OFF}
        {'H' FULL_ON}
        {'-' OFF}}
```

6.15 create_logic_net

Purpose	Define a logic net
Syntax	**create_logic_net** *net_name*
Arguments	*net_name* A simple name.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.

The **create_logic_net** command creates a logic net in the current scope or identifies a logic net in the current scope.

The net's type is determined by the language of the scope where it is created. If the scope is

— SystemVerilog, the type is `logic`

— VHDL, the type is `std_ulogic`

NOTE—This command exists to allow for the propagation of signals from a power-management block. Using this command to provide non-power control connections may cause the logic function to diverge from the HDL and is strongly discouraged.

Syntax example:

```
create_logic_net iso_ctrl
```

6.16 create_logic_port

Purpose	Define a logic port	
Syntax	**create_logic_port** *port_name* [**-direction** <**in** | **out** | **inout**>]	
Arguments	*port_name*	A simple name.
	-direction <**in** | **out** | **inout**>	The direction of the port. The default is **in**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_logic_port** command creates a logic port in the current scope. Logic ports are effectively created before isolation and level-shifting strategies are applied (see 4.3.3); therefore, any isolation or level-shifting strategy defined for a power domain may apply to logic ports created on the boundary of that power domain, regardless of the order in which the **create_logic_port** command and the **set_isolation** (see 6.41) or **set_level_shifter** (see 6.43) commands occur, provided the logic port matches the criteria specified in the strategy.

The port's type is determined by the language of the scope where it is created. If the scope is

— SystemVerilog, the type is `logic`

— VHDL, the type is `std_ulogic`

The created port is equivalent to a module port created in SystemVerilog or VHDL with the same name and direction. Logic ports are sources, sinks, or both.

a) The LowConn of an input port is a source.

b) The HighConn of an input port is a sink.

c) The LowConn of an output port is a sink.

d) The HighConn of an output port is a source.

e) The LowConn of an inout port is both a source and a sink.

f) The HighConn of an inout port is both a source and a sink.

NOTE—This command exists to allow for the propagation of signals from a power-management block. Using this command to provide non-power control connections may cause the logic function to diverge from the HDL and is strongly discouraged.

Syntax example:

```
create_logic_port test_lp
-direction out
```

6.17 create_power_domain

Purpose	Define a power domain and its characteristics		
Syntax	**create_power_domain** *domain_name* [**-simulation_only**] [**-atomic**] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-supply** {*supply_set_handle* [*supply_set_ref*]}]* [**-available_supplies** *supply_set_ref_list*] [**-define_func_type** {*supply_function pg_type_list*}]* [**-update**] [**-include_scope**] [**-scope** *instance_name*]		
Arguments	*domain_name*	The name of the power domain; this shall be a simple name rooted in the current scope.	
	-simulation_only	Define a power domain for simulation purposes only.	
	-atomic	Define the minimum extent of the power domain.	
	-elements *element_list*	The list of instances to add.	**R**
	-exclude_elements *exclude_list*	The list of instances to exclude from the *effective_element_list*.	**R**
	-supply {*supply_set_handle* [*supply_set_ref*]}	The **-supply** option specifies the *supply_set_handle* for *domain_name*. If *supply_set_ref* is also specified, the domain *supply_set_handle* is associated with the specified *supply_set_ref*. The *supply_set_ref* may be any supply set visible in the current scope. The predefined *supply_set_handle*s are: **primary**, **default_retention**, and **default_isolation**. See also 6.7.	**R**
	-available_supplies *supply_set_ref_list*	A list of additional supply sets that are available for use by implementation tools to power cells inserted in this domain.	
Arguments	**-define_func_type** {*supply_function pg_type_list*}	Define automatic connectivity for a *supply_function* of *domain_name*.**primary** (see 6.7) having the specified attributes in *pg_type_list*.	**R**
	-update	Use **-update** if the *domain_name* has already been defined.	**R**
Deprecated arguments	**-include_scope**	Define the extent of the domain to include the current scope and, by default, all of its descendant scopes. See also 5.8. This is a deprecated option; see also 6.1 and Annex D.	
	-scope *instance_name*	Create the power domain within this scope. This is a deprecated option; see also 6.1 and Annex D.	
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

create_power_domain defines a power domain and the set of instances that are in the extent of the power domain. It may also specify whether the power domain can be partitioned further by subsequent commands.

-elements specifies a set of rooted instances contained within the power domain. Although the syntax of this command does not include a **-transitive** option, its semantics are as if any occurrence of the command has the value **-transitive TRUE** (see 5.10.1). The following also apply:

- *element_list* shall contain instance names rooted in the current scope.
- Each design top instance (see 4.2.7) and each of its descendant instances shall be in the extent of exactly one power domain.
- When **-simulation_only** is specified, signal names and process labels may also be specified in *list*. **-simulation_only** specifies the domain is intended for use with behavioral non-synthesizing elements.
- When **-atomic** is specified, all elements originally included in the extent of the power domain shall always remain in the extent of that power domain.
- The power domain shall be created in the current scope.
- The **-elements** option shall be used at least once in the specification of a power domain using **create_power_domain**; this can be in the first invocation (i.e., without the **-update** option) or during the subsequent updates (i.e., with the **-update** option).
- If the value of *effective_element_list* (see 5.10) is an empty list, a domain with the name *domain_name* is created, but with an empty extent.
- If the value of the *effective_element_list* (see 5.10) is a period (`.`), the current scope is included in the extent of the domain.

NOTE 1—A design top instance can be included in the extent of a power domain created in the scope of that instance by specifying `-elements {.}` in the **create_power_domain** command.

NOTE 2—If the current scope is set to instance `i0`, then `create_power_domain PD -elements {.}` would include the current scope (`i0`) and all of its descendants in the power domain `PD`. In contrast, `create_power_domain PD -elements {i1 i2 ... ik}` would not include `i0` in the power domain, but would only include its descendants `i1`, `i2`, `...`, `ik`. In either case, the scope of the power domain `PD` is the same, because in both cases the current scope was `i0` when the **create_power_domain** command was executed.

An instance that has no parent or whose parent is in the extent of a different power domain is called a *boundary instance*.

The upper boundary of a power domain consists of

- the LowConn side of each port of each boundary instance in the extent of this domain.

The lower boundary of a power domain consists of

- the HighConn side of each port of each boundary instance in the extent of another power domain, where the parent of the boundary instance is in the extent of this domain, together with
- the HighConn side of each port of any macro cell instance in this power domain, for which the related supply set is neither identical to nor equivalent to the primary supply set of this domain.

The interface of a power domain consists of the union of the upper boundary and the lower boundary of the power domain.

create_power_domain also defines the supply sets that are used to provide power to instances within the extent of the power domain. The **-supply** option defines a supply set handle for a supply set used in the power domain.

A domain *supply_set_handle* may be defined without an association to a *supply_set_ref*. The association can be completed separately (see 6.7).

When both a *supply_set_handle* and a *supply_set_ref* are specified with **-supply**, the following supply set association is implied:

```
associate_supply_set supply_set_ref
   -handle domain_name.supply_set_handle
```

Three supply set handles are predefined for each power domain: **primary**, **default_isolation**, and **default_retention**.

The primary supply set is implicitly connected to instances and logic inferred from the instances in the power domain. However, the primary supply set shall not be implicitly connected when any of the following apply:

a) An instance has at least one supply net explicitly or automatically connected and **set_simstate_behavior** (see 6.53) has not been enabled.

b) An instance has **set_simstate_behavior** disabled.

c) An instance is created as a result of a UPF command, e.g., isolation cells, level-shifters, power switches, and retention registers.

Implicit connections imply simulation semantics as specified in 4.6.2.

The **default_isolation** supply set is the default supply for any isolation cell inserted into this domain if no isolation supply is specified in the **set_isolation** command (see 6.41). The applicable **default_isolation** supply is based upon the domain in which the isolation cell is inserted, not the domain for which the isolation strategy is defined.

The **default_retention** supply set is the default supply for any retention cell inserted into this domain if no retention supply is specified in the **set_retention** command (see 6.49).

Within a power domain, the predefined supply sets **primary**, **default_isolation**, and **default_retention** are available for use by implementation tools as required to power instances in the extent of the domain, isolation cells placed in the domain, and retention cells placed in the domain, respectively. Supply sets identified by command options of **set_isolation** (see 6.41), **set_level_shifter** (see 6.43), **set_repeater** (see 6.48), and **set_retention** (see 6.49) are also available to power isolation, level-shifter, repeater, and retention cells, respectively, inserted into the domain. Collectively, the predefined supply set handles of a power domain and the supply sets identified by options of strategies associated with the domain are referred to as the *locally available supplies* of that domain.

The **-available_supplies** option specifies whether any additional supplies are also available for use, and if so, which supplies are available. If **-available_supplies** does not appear, all supply sets and supply set handles defined in or above the scope of the power domain are available for use by tools to power cells inserted into the power domain. If **-available_supplies** appears with an empty string argument, only the locally available supplies are available for use by tools to power cells inserted into the power domain. If **-available_supplies** appears with a non-empty string, the string shall be a list of the names of additional supply sets or supply set handles defined at or above the scope of the power domain that are also available for use by tools to power cells inserted into the power domain, in addition to the locally available supplies.

Any restrictions on the availability of supply sets or supply set handles for use by tools to power cells inserted into a given domain have no effect on the legality of referencing such supply sets or supply set handles in UPF commands to associate supply sets with supply set handles or to connect supply set functions explicitly, implicitly, or automatically to supply pins of an instance.

-define_func_type specifies the mapping from functions of the domain's primary supply set to *pg_type* attribute values in the *pg_type_list*. This mapping determines the automatic connection semantics used to connect the domain's primary supply to instances within the extent of the domain.

-update may be used to add elements and supplies to a previously created domain. It shall be an error if **-update** is used during the initial creation of *domain_name*.

It shall be an error

- if an implementation tool encounters a **-simulation_only** power domain.
- for any instance in the descendant subtree of an atomic power domain to be included in the extent of another power domain, unless that instance name is, or is in the descendant subtree of, an instance whose name appears in the *exclude_list*.
- to remove an element from an atomic power domain.
- to specify **-atomic** with **-update**.
- to specify **-elements** or **-exclude_elements** with **-update** for an atomic power domain.

Syntax example:

```
create_power_domain PD1 -elements {top/U1}
-supply {primary}
-supply {default_isolation}
-supply {default_retention}
-supply {mem_array ss.mem}
create_power_domain PD2 -elements {.}
```

6.18 create_power_switch

Purpose	Define a power switch	
Syntax	**create_power_switch** *switch_name* **-output_supply_port** {*port_name* [*supply_net_name*]} {**-input_supply_port** {*port_name* [*supply_net_name*]}}* {**-control_port** {*port_name* [*net_name*]}}* {**-on_state** {*state_name input_supply_port* {*boolean_expression*}}}* [**-off_state** {*state_name* {*boolean_expression*}}]* [**-supply_set** *supply_set_ref*] [**-on_partial_state** {*state_name input_supply_port* {*boolean_expression*}}]* [**-ack_port** {*port_name net_name* [*logic_value*]}]* [**-ack_delay** {*port_name delay*}]* [**-error_state** {*state_name* {*boolean_expression*}}]* [**-domain** *domain_name*] [**-instance** {{*instance_name*}*}] [**-update**]	
Arguments	*switch_name*	The name of the switch instance to create; this shall be a simple name.
	-output_supply_port {*port_name* [*supply_net_name*]}	The output supply port of the switch and, optionally, the net where this port connects. *net_name* is a rooted name of a supply net or supply port. It shall be an error if the *net_name* is not defined in the current scope.

Arguments	**-input_supply_port** {*port_name* [*supply_net_name*]}	An input supply port of the switch and, optionally, the net where this port is connected. *net_name* is a rooted name of a supply net or supply port. It shall be an error if the *net_name* is not defined in the current scope.	
	-control_port {*port_name* [*net_name*]}	A control port on the switch and, optionally, the net where this control port connects. *net_name* is a rooted name of a logic net or logic port. It shall be an error if the *net_name* is not defined in the current scope.	
	-on_state {*state_name input_supply_port* {*boolean_expression*}}	A named on state, the *input_supply_port* for which this is defined, and its corresponding Boolean expression.	
	-off_state {*state_name* {*boolean_expression*}}	A named off state and its corresponding Boolean expression.	
	-supply_set *supply_set_ref*	A supply set associated with the switch. *supply_set_ref* is a rooted name of a supply set or a supply set handle. It shall be an error if the *supply_set_ref* is not defined in the current scope.	
	-on_partial_state {*state_name input_supply_port* {*boolean_expression*}}	A named partial-on state, the *input_supply_port* for which this is defined, and its corresponding Boolean expression.	
	-ack_port {*port_name net_name* [*logic_value*]}	The acknowledge port on the switch and the logic net to which this port connects. A logic value can also be specified. *net_name* is a rooted name of a logic net or logic port. It shall be an error if the *net_name* is not defined in the current scope. If a null string is used as the *net_name* for **-ack_port**, the port and its logic value are defined, but the port itself is unconnected.	
	-ack_delay {*port_name delay*}	The acknowledge delay for a given acknowledge port.	
	-error_state {*state_name* {*boolean_expression*}}	A named error state and its corresponding Boolean expression.	
	-domain *domain_name*	If specified, the scope of the domain is the scope in which the switch instance is created.	
	-instance {{*instance_name*}*}	The hierarchical name of a technology leaf-cell instance that implements all or part of the specified switch. Instance names are the hierarchical names of the switch instances.	**R**
	-update	Use **-update** to allow the addition of **-instance**.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **create_power_switch** command defines an abstract model of a power switch. An implementation may use detailed power-switching structures that involve multiple, distributed power switches in place of a single abstract power switch.

Power-switch port names and port state names are defined in the scope of the switch instance and, therefore, can be referenced with a hierarchical name in the same way that any other instance ports can be referenced. For example, the command

```
create_power_switch PS1
   -output_supply_port {outp}
   -input_supply_port {inp}
...
```

creates an instance `PS1` in the current scope and creates supply ports `outp` and `inp` within the `PS1` instance. The switch supply ports can then be referred to as `PS1/inp` and `PS1/outp`.

The abstract power-switch model has one or more input supply ports and one output supply port. Each of the input supplies may contribute to the output supply as determined by control expressions. Each input supply port is effectively gated by one or more control expressions defined by *on_state* or *on_partial_state* expressions. An *on_state* expression specifies when a given input supply contributes to the output without limiting current. An *on_partial_state* expression specifies when a given input supply contributes to the output in a current-limited manner. Each input supply may have multiple *on_state* and/or *on_partial_state* expressions.

The abstract power-switch model may also have one or more error_state expressions defined. Any *error_state* expressions defined for a given power switch represent control input conditions that are illegal for that switch.

The abstract power-switch model may also have a single off_state expression defined. The *off_state* expression represents the condition under which no *on_state* or *on_partial_state* expression is *True*. If not specified explicitly, the *off_state* expression defaults to the complement of the conjunction of all the *on_state*, *on_partial_state*, and *error_state* expressions defined for the power switch. It shall be an error if the *off_state* expression is explicitly defined and it evaluates to True when an *on_state* or *on_partial_state* expression also evaluates to *True*.

A contributing input supply port is one that has an *on_state* expression or *on_partial_state* expression that evaluates to *True* at a given time. The contributed value of a contributing input supply port is the value of the supply source connected to that input supply port. The degraded value of a contributing input supply port is the contributed value, except that if the contributed value's net state is **FULL_ON**, the degraded value's net state is **PARTIAL_ON**.

The value of the output supply port of a power switch is determined as follows. At any given time:

a) The output supply takes on the value {**UNDETERMINED**, unspecified} if
   1) any *error_state* condition is *True*, or
   2) an explicit *off_state* condition and any *on_state* or *on_partial_state* condition are both *True*, or
   3) any input supply port's contributed value has a net state of **UNDETERMINED**, or
   4) any two input supply ports' contributed values have different voltage values.

b) Otherwise, the switch output takes on the contributed value of any contributing input supply port whose net state is **FULL_ON**, if there is one.

c) Otherwise, the switch output takes on the degraded value of any contributing input supply port whose net state is **PARTIAL_ON**, if there is one.

d) Otherwise, the switch takes on the value {**OFF**, unspecified}.

An anonymous root supply driver originates the state of the output supply port when the state of the output supply port is explicitly set to **UNDETERMINED** or **OFF** in the preceding algorithm.

If an **–ack_port** argument is specified, an acknowledge value is driven onto the specified *port_name delay* time units after the switch output transitions to a **FULL_ON** state and the inverse acknowledge value is driven onto the specified *port_name delay* time units after the switch output transitions to an **OFF** state.

If the supply set of the power switch is in a power state with a **NORMAL** simstate, then the acknowledge value is a logic `0` or logic `1`. If a *logic_value* is specified for **-ack_port**, that logic value shall be used as the acknowledge value for a transition to **FULL_ON**, and its negation is used as the acknowledge value for a transition to **OFF**; otherwise the acknowledge value defaults to logic `1` for a transition to **FULL_ON** and logic `0` for a transition to **OFF**. If **-ack_delay** is specified, the delay may be specified as a time unit, or it may be specified as a natural integer, in which case the time unit shall be the same as the simulation precision; otherwise, the delay defaults to `0`.

If **-supply_set** is specified for a switch, it powers logic or timing control circuitry within the switch and powers any specified **-ack_port**s. When the supply set simstate is anything other than **NORMAL**, the state of the output supply port of a switch is **UNDETERMINED** and the acknowledge ports are corrupted. If a supply set is not associated with a switch, it shall be an error if any acknowledge ports are specified.

-instance specifies that the power-switch functionality exists in the HDL design and *instance_name* denotes the instance providing part or all of this functionality. If **-instance** is specified, and a list of instances is given, then the switch may be implemented as multiple switches, in which case the multiple instances may have characteristics different from those specified by the **create_power_switch** command, particularly with regard to input and output supply connections.

An *instance_name* is a hierarchical name rooted in the current scope. If an empty string appears in an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance should not be reinferred or reimplemented by subsequent tool runs.

Updating **–instance** adds the new instance names to the existing instance list. **–update** adds information to the base command executed in the same scope in which the object exists or is to be created.

The following also apply:

- Any name in a *boolean_expression* shall refer to a control port of the switch.
- All states not covered by the on, on_partial, off, and error states are anonymous error states.
- If the implementation of a switch can not be inferred, **map_power_switch** (see 6.32) can be used to specify it.
- If *net_name* is not specified for any of the switch's port definitions, **connect_logic_net** (see 6.10) or **connect_supply_net** (see 6.11) can be used to create the port connections.
- Each state name shall be unique for a particular switch.
- Any *port_name*s specified in this command are user defined (e.g., `input_supply`).

NOTE 1—**create_power_switch** can be used to define an abstract power switch that implementation tools may expand into multiple switches. **create_power_switch** can also be used to specify the need for a specific switch that can then be mapped to a specific switch implementation using **map_power_switch**. It is not meant to be used as a single definition representing multiple physical switches to be mapped with **map_power_switch**.

NOTE 2—**create_power_switch** provides relatively simple, general abstract functionality. HDLs can be used to model switch functionality that cannot be captured with **create_power_switch**.

Power-switch examples

Example 1—Simple switch

This switch model has a single supply input and a single control input. The switch is either on or off, based on the control input value. Since net names are not specified for each port, **connect_supply_net** (see 6.11) can be used to connect a net to each port.

```
create_power_switch simple_switch
-output_supply_port {vout}
-input_supply_port {vin}
```

```
-control_port {ss_ctrl}
-on_state {ss_on vin { ss_ctrl }}
-off_state {ss_off { ! ss_ctrl }}
```

The following is a variant of the simple switch in which the nets associated with the ports are defined as part of the **create_power_switch** command (see 6.18).

```
create_power_switch simple_switch2
-output_supply_port {vout VDD_SW}
-input_supply_port {vin VDD}
-control_port {ss_ctrl sw_ena}
-on_state {ss_on vin { ss_ctrl }}
-off_state {ss_off { ! ss_ctrl }}
```

Example 2—Two-stage switch

This switch model represents a switch that turns on in two stages. The switch has one supply input and two control inputs. One control input represents the enable for the first stage; the other represents the control for the second stage. When only the first control is on, the switch output is in a partial on state; when the second is on, the switch output is in a fully on state. The switch is off if neither control input is on.

```
create_power_switch two_stage_switch
-output_supply_port {vout}
-input_supply_port {vin}
-control_port {trickle_ctrl}
-control_port {main_ctrl}
-on_partial_state {ts_ton vin { trickle_ctrl }}
-on _state {ts_mon vin { main_ctrl }}
-off_state {ts_off { ! trickle_ctrl && ! main_ctrl }}
```

The following is a variant of the two-stage switch model in which an **-ack_port** signals completion of the switch turning on. The time required for the switch to turn on is modeled by the **-ack_delay**. Since an **-ack_port** is involved, the command needs to include specification of the supply set that powers the logic driving the ack signal. The ack signal is defined separately. In this model, as in the preceding simple switch variant, the supply and control ports are associated with corresponding nets, so they do not need to be connected in a separate step.

```
create_power_switch two_stage_switch2
-output_supply_port {vout VDD_SW}
-input_supply_port {vin VDD}
-control_port {trickle_ctrl t_ena}
-control_port {main_ctrl m_ena}
-on_partial_state {ts_ton vin { trickle_ctrl }}
-on _state {ts_mon vin { main_ctrl }}
-off_state {ts_off { ! trickle_ctrl && ! main_ctrl }}
-ack_port {ts_ack 1}
-ack_delay {ts_ack 100ns}
-supply_set ss_aon
```

Example 3—Muxed switch

This switch model represents a mux that determines which of two different input supplies is connected to the output supply port at any given time. The two input supplies can be driven by different root supply drivers and may have different state/voltage values. One control input determines which of the two input supplies is selected; the other control input gates the selected supply to the output supply.

```
create_power_switch muxed_switch
-output_supply_port {vout}
-input_supply_port {vin0}
-input_supply_port {vin1}
-control_port {ms_sel}
-control_port {ms_ctrl}
-on_state {ms_on0 vin0 { ms_ctrl && ! ms_sel }}
-on_state {ms_on1 vin1 { ms_ctrl && ms_sel }}
-off_state {ms_off { ! ms_ctrl }}
```

The following is a variant of the muxed switch in which there are two independent selection control inputs, and an error state is defined to ensure mutual exclusion.

```
create_power_switch muxed_switch2
-output_supply_port {vout}
-input_supply_port {vin0}
-input_supply_port {vin1}
-control_port {ms_sel0}
-control_port {ms_sel1}
-control_port {ms_ctrl}
-on_state {ms_on0 vin0 { ms_ctrl && ms_sel0 }}
-on_state {ms_on1 vin1 { ms_ctrl && ms_sel1 }}
-off_state {ms_off { ! ms_ctrl }}
-error_state {conflict { ms_sel0 && ms_sel1 }}
```

Example 4—Overlapping muxed switch

This switch model represents a supply mixer that allows a smooth transition between two different supplies. Like the muxed switch, it has two supply inputs and both selecting and gating control inputs, but in this case it can select both input supplies at the same time. The input supplies may have different states, and may even be driven by different root supply drivers, provided that their voltages are the same when both inputs are enabled (in an on state or on_partial state).

```
create_power_switch overlapping_muxed_switch
-output_supply_port {vout}
-input_supply_port {vin0}
-input_supply_port {vin1}
-control_port {oms_sel0}
-control_port {oms_sel1}
-control_port {oms_ctrl}
-on_state {oms_on0 vin0 { oms_ctrl && oms_sel0 }}
-on_state {oms_on1 vin1 { oms_ctrl && oms_sel1 }}
-off_state {oms_off { !oms_ctrl || { !oms_sel0 && !oms_sel1 } }}
```

6.19 create_pst [legacy]

Purpose	Create a power state table (PST)	
Syntax	**create_pst** *table_name* **-supplies** *supply_list*	
Arguments	*table_name*	The PST name. *table_name* is a simple name in the current scope.
	-supplies *supply_list*	The list of supply nets or ports to include in each power state of the design. The supplies are listed as rooted names in the current scope.
Return value	Return the name of the created PST or raise a `TCL_ERROR` if the PST is not created.	

This is a legacy command; see also 6.1 and Annex D.

The **create_pst** command defines a PST name and a set of supply nets for use in **add_pst_state** commands (see 6.5). The PST *table_name* is defined in the namespace of the current scope.

A PST is used for implementation—specifically for synthesis, analysis, and optimization. It defines the legal combinations of states, i.e., those combinations of states that can exist at the same time during operation of the design.

create_pst can only be used with **add_pst_state** (and vice versa). This combination and use of **add_power_state** (see 6.4) are two methods for specifying power state information. Power state specifications and default state definitions form an exhaustive specification of all of the legal power states of the system.

It shall be an error if

— *table_name* conflicts with any name declared in the namespace of the current scope.

— a specified supply net or supply port specified in *supply_list* does not exist.

Syntax example:

```
create_pst MyPowerStateTable -supplies {PN1 PN2 SOC/OTC/PN3}
```

6.20 create_supply_net

Purpose	Create a supply net	
Syntax	**create_supply_net** *net_name* [**-domain** *domain_name*][**-reuse**] [**-resolve** <**unresolved** | **one_hot** | **parallel** | **parallel_one_hot**>]	
Arguments	*net_name*	A simple name.
	-domain *domain_name*	The domain in whose scope the supply net is to be created.
	-reuse	Extend availability of a supply net previously defined for another domain into this domain.

Arguments	**-resolve** <**unresolved** | **one_hot** | **parallel** | **parallel_one_hot**>	A resolution mechanism that determines the state and voltage of the supply net when the net has multiple supply sources (see 6.20.2). If no option is specified, the behavior for resolution is the same as for **unresolved**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_supply_net** command creates a supply net. If **-domain** is not specified, the supply net is created in the current scope, and the supply net is available for use by tools to power cells in any domain created at or below this scope. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant tree of the scope in which the net is created as required by implicit and automatic connections of supply sets (see 6.17).

If **-domain** is specified, the supply net is created in the scope of that domain, and the supply net is available for use by tools to power cells only in the extent of the domain.

If **–reuse** is specified, a supply net with this name needs to have been created by a previously executed command, and this existing supply net is made available for use in another domain by the **–domain** option. In this case:

a) **-domain** shall also be specified on both this and the creating command;
b) **-resolve** shall not conflict with that of the creating command.

The following also apply:

— It shall be an error if *domain_name* is not the name of a previously created power domain.
— When **-reuse** is specified, it shall be an error if *net_name* is not defined for another power domain in the same scope by another **create_supply_net** command.
— When the parameter for **-resolve** is **unresolved**, the supply net shall have only one source (see 6.20.1). For all other parameters to **-resolve**, the requirements on the drivers and sources of the net are as defined in 6.20.2.

NOTE—Use **set_scope** (see 6.52) to change the scope prior to calling this command to set the current scope to the correct scope for the net.

Syntax example:

```
create_supply_net local_vdd_3
   -resolve one_hot
```

6.20.1 Supply net resolution

Supply nets are often connected to the output of a single switch. However, certain applications, such as on-chip voltage scaling, may require the outputs of multiple switches or other supply drivers to be connected to the same supply net (either directly or via supply port connections). In these cases, a resolution mechanism is needed to determine the state and voltage of the supply net from the state and voltage values supplied by each of the supply drivers to which the net is connected.

A supply net that specifies an **unresolved** resolution cannot be connected to more than one supply source.

6.20.2 Resolutions methods

The semantics of each possible resolution method are as follows:

a) unresolved

The supply net shall be connected to at most one supply source. This is the default.

b) one_hot

Multiple supply sources, each having a unique driver, may be connected to the supply net.

A supply net with **one_hot** resolution has a deterministic state only when no more than one source drives the net at any particular point in time. If at any point in time more than one supply source driving the net is anything other than **OFF**, the state of the supply net shall be **UNDETERMINED**, the voltage value of the supply net shall be unspecified, and implementations may issue a warning or an error.

1) If all supply sources are **OFF**, the state of the supply net shall be **OFF**, and the voltage value of the supply net shall be unspecified.
2) If only one supply source is **FULL_ON** and all other sources are **OFF**, the state of the supply net shall be **FULL_ON**, and the voltage value of the corresponding source shall be assigned to the supply net.
3) If only one supply source is **PARTIAL_ON** and all other sources are **OFF**, the state of the supply net shall be **PARTIAL_ON** and the voltage value of the corresponding source shall be assigned to the supply net.
4) If any source is **UNDETERMINED**, the state of the supply net shall be **UNDETERMINED**, and the voltage value of the supply net shall be unspecified.

c) parallel

Multiple supply sources, sharing a common root supply driver, may be connected to the supply net.

The **parallel** resolution allows more than one potentially conducting path to the same root supply driver, as if the switches had been connected in parallel. It shall be an error if any of these potentially conducting paths can be traced to more than one root supply driver.

1) If all of the supply sources are **FULL_ON**, then the supply net state is **FULL_ON** and the voltage value is the value of the root supply driver.
2) If all the supply sources driving the supply net are **OFF**, the state of the supply net shall be **OFF** and the voltage is unspecified.
3) If any of the sources is **UNDETERMINED**, the resolution is **UNDETERMINED**; otherwise,
    i) If there is at least one **PARTIAL_ON** source, the supply net shall be **PARTIAL_ON** and the voltage value is the value of the root supply driver.
    ii) If there is at least one source that is **OFF** and at least one that is **FULL_ON** or **PARTIAL_ON**, the supply net shall be **PARTIAL_ON** and the voltage value is the value of the root supply driver. The voltage value of the **PARTIAL_ON** supply net shall be the voltage value of the root supply driver.

d) parallel_one_hot

Multiple supply sources may be connected to the supply net. A source may share a common root supply driver with one or more other sources. At most one root supply driver shall be **FULL_ON** at any particular time with all sources sharing that driver resolved using parallel resolution.

The **parallel_one_hot** resolution allows resolution of a supply net that has multiple root supply drivers where each driver may have more than one path through supply sources to the supply net. Each unique root supply driver is identified and **one_hot** resolution shall be applied to the drivers, then **parallel** resolution shall be applied to each supply source connecting the **one_hot** root supply driver to the supply net.

6.20.3 Supply nets defined in HDL

The declaration of any VHDL `signal` or SystemVerilog `wire` or `reg` as a `supply_net_type` from the package UPF (see Annex B) is equivalent to calling **create_supply_net** for every instance of that declaration, where the *net_name* is the name of the VHDL `signal` or SystemVerilog `wire` or `reg`, and the scope is the instance.

6.21 create_supply_port

Purpose	Create a supply port on a instance	
Syntax	**create_supply_port** *port_name* [**-domain** *domain_name*] [**-direction** <**in** | **out** | **inout**>]	
Arguments	*port_name*	A simple name.
	-domain *domain_name*	The domain where this port defines a supply net connection point.
	-direction <**in** | **out** | **inout**>	The direction of the port. The default is **in**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_supply_port** command defines a supply port at the scope of the power domain when **-domain** is specified or at the current scope if **-domain** is not specified.

-direction defines how state information is propagated through the supply network as it is connected to the port. If the port is an input port, the state information of the external supply net (see 6.20) connected to the port shall be propagated into the instance. Likewise, for an output port, the state information of the internal supply net connected to the port shall be propagated outside the instance.

Supply ports connected to a net shall be **inout** for supply nets that have both loads and sources within that module. Supply ports are loads, sources, or both, as follows:

a) The LowConn of an input port is a source.
b) The HighConn of an input port is a sink.
c) The LowConn of an output port is a sink.
d) The HighConn of an output port is a source.
e) The LowConn of an inout port is both a source and a sink.
f) The HighConn of an inout port is both a source and a sink.

Supply ports may be defined in HDL. If a VHDL or SystemVerilog `port` is declared as a `supply_net_type` from the package UPF (see Annex B), this is equivalent to calling **create_supply_port** for every instance of that declaration, where the *port_name* is the name of the VHDL or SystemVerilog `port`, and the scope is the instance.

Syntax example:

```
create_supply_port VN1
   -direction inout
```

6.22 create_supply_set

Purpose	Create or update a supply set, or update a supply set handle		
Syntax	**create_supply_set** *set_name* [**-function** {*func_name net_name*}]* [**-reference_gnd** *supply_net_name*] [**-update**]		
Arguments	*set_name*	The simple name of the supply set or a supply set handle.	
	-function {*func_name net_name*}	The **-function** option defines the function (*func_name*) a supply net provides for this supply set. *net_name* is a rooted name of a supply net or supply port or a supply net handle. It shall be an error if the net_name is not defined in the current scope.	**R**
	-reference_gnd *supply_net_name*	The **-reference_gnd** option defines the rooted name of a *supply_net* that serves as the reference ground for the supply set. A supply net handle may be used. **Default**: if not specified, the voltages in this supply set shall be evaluated with no offset from the assumed default reference supply.	**R**
	-update	Use **-update** if the *set_name* has already been defined.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

create_supply_set creates the supply set name within the current scope in the UPF name space. The reference ground can be specified in any invocation of this command. This command defines a *supply set* as a collection of supply nets each of which serve a specific function for the set.

-update is used to signify that this **create_supply_set** call refers to a supply set that was previously defined using **create_supply_set**, or to a supply set handle that was previously defined implicitly or explicitly using **create_power_domain** (see 6.17). Referencing a previously created supply set or supply set handle without the **-update** argument shall be an error. Using the **-update** argument for a supply set that has not been previously defined shall be an error. Specifying a supply set handle that has not been previously defined shall be an error.

When **-function** is specified, *func_name* shall be one of the following: **power**, **ground**, **nwell**, **pwell**, **deepnwell**, and **deeppwell**. The **–function** option associates the specified *func_name* of this supply set with the specified *supply_net_name*. If the same *func_name* is associated with two different supply nets, it shall be an error if those supply nets are not the same. The *supply_net_name* may be a reference to a supply net in the descendant hierarchy of the current scope using a supply net handle (see 6.22.1).

When **-reference_gnd** is specified, *supply_net_name* is the name of a supply net that serves as the reference ground for the supply set. The voltage value for each supply net in the supply set is interpreted in reference to this supply net. If this parameter is not specified, the voltages shall be evaluated with no offset or scaling. If **-reference_gnd** has previously had a *supply_net_name* specified, then it shall be an error if this *supply_net_name* and the *supply_net_name* previously specified as reference ground are not equivalent nets.

Syntax example:

```
create_supply_set relative_always_on_ss
  -function {power vdd}
  -function {ground vss}
```

```
create_supply_set relative_always_on_ss -update
  -reference_gnd {earth_ground}
create_supply_set PD1.primary -update
      -function {nwell bias}
```

6.22.1 Referencing supply set functions

The supply set function may also be referenced using a supply net handle as a member of the supply set (whether or not a supply net has been associated with the function name), as follows:

supply_set_ref.function

6.22.2 Implicit supply net

If no supply net is associated with a supply set's function and that function is used in the design, an implicit supply net with an anonymous name shall be created for use in verification and analysis. When the UPF specification is used for implementation, a supply net shall not be implicitly created for a supply set function that has no associated supply net. A tool may issue a warning or an error if a supply set's function does not have an explicit supply net association.

6.23 create_upf2hdl_vct

Purpose	Define VCT that can be used in converting UPF `supply_net_type` values into HDL logic values	
Syntax	**create_upf2hdl_vct** *vct_name* **-hdl_type** {<**vhdl** | **sv**> [*typename*]} **-table** {{*from_value to_value*}*}	
Arguments	*vct_name*	The VCT name.
	-hdl_type {<**vhdl** | **sv**> [*typename*]}	The HDL type for which the value conversions are defined.
	-table {{*from_value to_value*}*}	A list of UPF `state` type values to map to the values of the HDL type.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **create_upf2hdl_vct** command defines a value conversion table (VCT) for the `supply_net_type.state` value (see Annex B) when that value is propagated from a UPF supply net into a logic port defined in an HDL. It provides a 1:1 conversion for each possible combination of the partially on and on/off states. **create_upf2hdl_vct** does not check that the values are compatible with any HDL port type.

vct_name provides a name for the value conversion table for later use with the **connect_supply_net** command (see 6.11). The predefined VCTs are shown in Annex F.

-hdl_type specifies the HDL type for which the value conversions are defined. This information allows a tool to provide completeness and compatibility checks. If the *typename* is not one of the language's predefined types or one of the types specified in the next paragraph, then it shall be of the form *library*`.`*pkg*`.`*type*.

The following HDL types shall be the minimum set of types supported. An implementation tool may support additional HDL types.

a) VHDL
   1) `Bit`, `std_[u]logic`, `Boolean`
   2) Subtypes of `std_[u]logic`
b) SystemVerilog
   `reg/wire`, `Bit`, `Logic`

-table defines the 1:1 conversions from UPF supply net states to an HDL logic value. The values shall be consistent with the HDL type values. For example:

— When converting to SystemVerilog *logic type*, the set of legal values is `0`, `1`, `X`, and `Z`.
— When converting to SystemVerilog or VHDL `bit`, the legal values are `0` or `1`.
— When converting to VHDL `std_[u]logic`, the legal values are `U`, `X`, `0`, `1`, `Z`, `W`, `L`, `H`, and `-`.

The conversion values have no semantic meaning in UPF. The meaning of the conversion value is relevant to the HDL model to which the supply net is connected.

Syntax examples:

```
create_upf2hdl_vct upf2vlog_vdd
  -hdl_type {sv}
  -table {{OFF X} {FULL_ON 1} {PARTIAL_ON 0}}
create_upf2hdl_vct upf2vhdl_vss
  -hdl_type {vhdl std_logic}
  -table {{OFF 'X'} {FULL_ON '1'} {PARTIAL_ON 'H'}}
```

6.24 describe_state_transition

Purpose	Describe a state transition's legality	
Syntax	**describe_state_transition** *transition_name* **-object** *object_name* [**-from** *from_list* **-to** *to_list*] [**-paired** {{*from_state to_state*}*}] [**-legal** | **-illegal**]	
Arguments	*transition_name*	Simple name.
	-object *object_name*	Simple name of a power domain or supply set.
	-from *from_list* **-to** *to_list*	*from_list* is an unordered list of power state names active before a state transition. *to_list* is an unordered list of power state names active after a state transition.
	-paired {{*from_state to_state*}*}	A list of from-state name and to-state name pairs.
	-legal | **-illegal**	Define the state transition as legal or illegal, the default is **-legal**.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

describe_state_transition specifies the legality of a transition from one object's named power state to another.

The option **-from** and **-to** specify many-to-many transitions. The option **-paired** specifies one or more one-to-one transitions. At least one of these two choices shall be specified.

If an empty list is specified in either the **-from** or **-to** *list*, it shall be expanded to all named power states for the specified *object_name*.

Verification tools shall emit an error when an illegal state transition occurs.

It shall be an error if the state name in a *list* does not refer to a power state of the specified supply state or power domain (see 6.4).

Syntax example:

```
describe_state_transition turn_on -object PdA -from {SLEEP_MODE}
   -to {HIGH_SPEED_MODE} -illegal
```

6.25 end_power_model

Purpose	Terminate the definition of a power model
Syntax	**end_power_model**
Arguments	N/A
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.

The **begin_power_model** (see 6.8) and **end_power_model** commands define a power model containing other UPF commands. A power model is used to define the power intent of a hard IP and shall be used in conjunction with one or more model representations. A power model defined with **begin_power_model** is terminated by the first subsequent occurrence of **end_power_model** in the same UPF file.

6.26 find_objects

Purpose	Find logic hierarchy objects within a scope	
Syntax	**find_objects** *scope* **-pattern** *search_pattern* [**-object_type** <**model** | **inst** | **port** | **net** | **process**>] | [**-direction** <**in** | **out** | **inout**>] [**-transitive** [<**TRUE** | **FALSE**>]] [**-regexp** | **-exact**] [**-ignore_case**] [**-non_leaf** | **-leaf_only**]	
Arguments	*scope*	The search is restricted to the specified scope.
	-pattern *search_pattern*	The string used for searching. By default, *search_pattern* is treated as an Tcl `glob` expression.
	-object_type <**model** | **inst** | **port** | **net** | **process**>	Limits the objects returned. By default, instances, named processes, ports, and nets are returned; this can be restricted by specifying a specific **-object_type**. **inst** does not return named processes.
	-direction <**in** | **out** | **inout**>	If **-object_type** is **port**, then **-direction** can be used to restrict the directions of the returned ports.
	-transitive [<**TRUE** | **FALSE**>]	If **-transitive** is not specified at all, the default is **-transitive FALSE**. If **-transitive** is specified without a value, the default value is **TRUE**.
	-regexp | **-exact**	**-regexp** enables support for regular expression in the specified *search_pattern*. **-exact** disallows wildcard expansion on the specified *search_pattern*. If neither **-regexp** or **-exact** are specified, then *search_pattern* is interpreted as a Tcl `glob` expression.
	-ignore_case	Performs case-insensitive searches. By default, all matches are case sensitive.
	-non_leaf | **-leaf_only**	If **-non_leaf** is specified, only non-leaf instances (instances that have children) are returned; if **-leaf_only** is specified, only leaf-level instances (instances without children) are returned. By default, both leaf and non-leaf instances are returned.
Return value	Returns a list of names (relative to the current scope) of objects that match the search criteria; when nothing is found that matches the search criteria, a *null string* is returned. The list contains just the object names, without any indication of object type. The list may contain names of more than one type of object.	

The **find_objects** command searches for instances, nets, ports, or processes that are defined in the logic hierarchy. If **-object_type model** is specified, **find_objects** searches for instances of any model whose model name matches the search_pattern. If **-object_type** is specified with any other value, **find_objects** searches the logic hierarchy for the specified objects whose name matches the *search_pattern*.

By default, or if **-transitive FALSE** is specified explicitly, **find_objects** searches only the current scope of the logic hierarchy. If **-transitive TRUE** is specified, **find_objects** searches the current scope and the entire dependant subtree. If **-transitive** is specified without an argument, it is equivalent to specifying **-transitive TRUE**.

NOTE—To find UPF objects, such as isolation logic or retention elements, use the corresponding **query_*** commands (see Annex C).

The **-non_leaf** and **-leaf_only** options can be interpreted differently between tools, depending upon the library source. For example, in simulation, an IP block may be represented as a non-leaf hierarchical behavioral model, whereas in implementation, the same IP block may be represented as a black box leaf cell. A module may be tagged as a leaf cell by using **set_design_attributes** (see 6.37).

The following conditions also apply:

- The specified *scope* cannot start with `..` or `/`, i.e., **find_objects** shall be referenced from the current scope, and reside in the current scope or below it.
- If *scope* is specified as **.** (a dot), the current scope is used as the root of the search.
- All elements returned are referenced to the current scope.
- It shall be an error if *scope* is neither the current scope nor is defined in the current scope. The specified scope may reference a generate block as the root of the search.
- While **find_objects** commands are executed and their results are used; the command itself is not saved. However, this does not prohibit the use of **find_objects** in output UPF.

Syntax examples:

```
find_objects A/B/D -pattern *BW1*
   -object_type inst
   -transitive TRUE
```

6.26.1 Pattern matching and wildcarding

To improve usability and allow multiple objects (instances, ports, etc.) to be easily specified without onerous verbosity, pattern matching (wildcarding) is allowed (only) in **find_objects** and **query_upf** (see C.1). Pattern matching is supported using the Tcl `glob` style, matching against the symbols in the scope rather than filenames. For `glob`-style wildcarding, the following special operators are supported:

? matches any single character.

***** matches any sequence of zero or more characters.

[*chars***]** matches any single character in *chars*. If *chars* contains a sequence of the form `a-b`, any character between `a` and `b` (inclusive) shall match.

x matches the character *x*.

{*a*,*b*,*c*} Matches any string that is matched by any of the patterns *a*, *b*, or *c*.

Tcl regular expression matching is described in the Tcl documentation for `re_syntax` (see [B5]).

6.26.2 Wildcarding examples

Table 5 shows the pattern match for each of the following examples of **find_objects**.

```
find_objects top -pattern a
find_objects top -pattern {bc[0-3]}
find_objects top -pattern e*
find_objects top -pattern d?f
find_objects top -pattern {g\[0\]}
```

NOTE 1—The use of the Tcl quote semantics of "{*string*}" in the example illustrates an effective means to pass characters that would otherwise be "special" to a Tcl interpreter.

NOTE 2—To select the four bits (`0` to `3`) of the bus `my_bus`, use the Tcl expression `{my_bus\[[0-3]\]}`.

Table 5—Pattern matches

`a`	Only matches an instance called `a` in the current scope.
`bc[0-3]`	Matches any instance called `bc` followed by a numerical value from `0` to `3`, i.e., `bc0`, `bc1`, `bc2`, and `bc3`.
`e*`	Matches any instance starting with `e`, i.e., `e12`, `eab`, `ef`, etc.
`d?f`	Matches any instance starting with `d` followed by another character and ending in `f`, i.e., `daf`, `d4f`, etc.
`g\[0\]`	Matches an instance called `g[0]`.

6.27 load_simstate_behavior

Purpose	Load the simstate behavior defaults for a library	
Syntax	**load_simstate_behavior** *lib_name* **-file** *file_list*	
Arguments	*lib_name*	The tool specific library name for which the simstate behavior file is to be loaded.
	-file *file_list*	The list of files containing the **set_simstate_behavior** commands.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

Loads a UPF file that only contains **set_simstate_behavior** commands and applies these to the models in the library *lib_name*.

It shall be an error if

- *lib_name* cannot be resolved.
- *file_list* does not exist.
- a model specified in *file_list* cannot be found.
- the **set_simstate_behavior** commands in *file_list* use the **-lib** argument.
- *file_list* contains UPF commands other than **set_simstate_behavior.**

Syntax example:

```
load_simstate_behavior library1 -file simstate_file.upf
```

6.28 load_upf

Purpose	Set the scope to the specified instance and execute the specified UPF commands	
Syntax	**load_upf** *upf_file_name* [**-scope** *instance_name_list*] [**-version** *upf_version*]	
Arguments	*upf_file_name*	The UPF file to execute.
	-scope *instance_name_list*	The list of scopes where the UPF commands contained in *upf_file_name* are executed.
	-version *upf_version*	The UPF version for which commands in this file are written. See also 6.54.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **load_upf** command sets the scope to each of the specified list of instances and executes the set of UPF commands in the file *upf_file_name*. Upon return, the current scope is restored to what it was prior to invocation. If a scope specified in *instance_name_list* is not found, further processing of remaining scopes in the *instance_name_list* is terminated and a `TCL_ERROR` is raised.

load_upf does not create a new name space for the loaded UPF file. The loaded UPF file is responsible for ensuring the integrity of both its own and the caller's name space as needed using existing Tcl name space management capabilities.

If **-scope** is specified, each instance name in the instance name list shall be a simple name or a hierarchical name rooted in the current scope. In this case, for the duration of the **load_upf** command, the current scope and design top instance are both set to the instance specified by the instance name and the design top module is set to the module type of that instance.

When the **load_upf** command completes, the current scope, design top instance, and design top module all revert to their previous values.

If –**version** *upf_version* is specified, the command

```
upf_version upf_version
```

is implicitly executed before executing the commands in the loaded file.

Syntax example:

```
load_upf my.upf -scope {I1/I2 I3/I2} -version 2.1
```

6.29 load_upf_protected

Purpose	Load a UPF file in a protected environment that prevents corruption of existing variables	
Syntax	**load_upf_protected** *upf_file_name* [**-hide_globals**] [**-scope** *instance_name_list*] [**-version** *upf_version*] [**-params** *param_list*]	
Arguments	*upf_file_name*	The UPF file to be sourced.
	-hide_globals	Save all globals before sourcing *upf_file_name* and restore them afterwards. Globals named in the *param_list* retain any modified values resulting from sourcing the file. Any globals not in the *param_list* shall be unset before *upf_file_name* is loaded. Any globals created in the sourced file, other than the ones named in *param_list*, are unset at the end of loading.
	-scope *instance_name_list*	The list of scopes where the UPF commands contained in *upf_file_name* are executed.
	-version *upf_version*	The UPF version for which commands in this file are written. See also 6.54.
	-params *param_list*	A list of variables to be made available while sourcing the file. In *param_list*, each element has one of the following formats: a) *param_name* — declared as `"global $paramName"`. Any changes made to this variable are visible at the calling level once this command completes. b) {*param_name param_value*} — a local variable *param_name* is created and its initial value is set to *param_value*. The Tcl variable `errorInfo` shall behave as if it has been specified in this list.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

If a scope specified in *instance_name_list* is not found, further processing of remaining scopes in the *instance_name_list* is terminated and a `TCL_ERROR` is raised.

If **-scope** is specified, each instance name in the instance name list shall be a simple name or a hierarchical name rooted in the current scope. In this case, for the duration of the **load_upf_protected** command, the current scope and design top instance are both set to the instance specified by the instance name and the design top module is set to the module type of that instance.

When the **load_upf_protected** command completes, the current scope, design top instance, and design top module all revert to their previous values.

If –**version** *upf_version* is specified, the command

```
upf_version upf_version
```

is implicitly executed before executing the commands in the loaded file.

Syntax example:

```
load_upf_protected my.upf -hide_globals -version 2.0
```

6.30 map_isolation_cell [deprecated]

This is a deprecated command; see also 6.1 and Annex D.

6.31 map_level_shifter_cell [deprecated]

This is a deprecated command; see also 6.1 and Annex D.

6.32 map_power_switch

Purpose	Specify which power-switch model is to be used for the implementation of the corresponding switch instance	
Syntax	**map_power_switch** *switch_name_list* **-lib_cells** *lib_cells_list* [**-port_map** {{*mapped_model_port switch_port_or_supply_net_ref*}*}] [**-domain** *domain_name*]	
Arguments	*switch_name_list*	A list of switches [as defined by **create_power_switch** (see 6.18)] to map.
	-lib_cells *lib_cells_list*	A list of library cells.
	-port_map {{*mapped_model_port switch_port_or_supply_net_ref*}*}	*mapped_model_port* is a port on the model being mapped. *switch_port_or_supply_net_ref* indicates a supply or logic port on a switch: an input supply port, output supply port, control port, or acknowledge port; or it references a supply net from a supply set associated with the switch. See also **create_power_switch** (6.18) or **set_power_switch** (6.47).
Deprecated arguments	**-domain** *domain_name*	This argument is ignored and provided for syntactic backward compatibility only. This is a deprecated option; see also 6.1 and Annex D
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **map_power_switch** command can be used to explicitly specify which power-switch model is to be used for the corresponding switch instance.

-lib_cells specifies the set of library cells to which an implementation can be mapped. Each cell specified in **-lib_cells** shall be defined by a **define_power_switch_cell** command (see 7.6) or defined in the Liberty file with required attributes.

If **-port_map** is not specified, the ports of the switch instance are associated to library cell ports by matching the respective port names, this is *named association*. It shall be an error if any ports on either the switch instance or the library cell are not mapped when named association is used.

It shall be an error if *switch_name_list* is an empty list.

NOTE 1—All **map_*** commands specify the elements to be used rather than inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, logical equivalence checking tools may not be able to verify the equivalence of the mapped element to its RTL specification.

NOTE 2—**create_power_switch** can be used to define an abstract power switch that implementation tools may expand into multiple switches. **create_power_switch** can also be used to specify the need for a specific switch that can then be mapped to a specific switch implementation using **map_power_switch**. It is not meant to be used as a single definition representing multiple physical switches to be mapped with **map_power_switch**.

Syntax example:

```
map_power_switch switch_sw1
-domain test_suite
-lib_cells {sw1}
-port_map {{inp1 vin1} {inp2 vin2} {outp vout}
   {c1 ctrl_small} {c2 ctrl_large}}
```

6.33 map_retention_cell

Purpose	Constrain implementation alternatives, or specify a functional model, for retention strategies	
Syntax	**map_retention_cell** *retention_name_list* **-domain** *domain_name* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-lib_cells** *lib_cell_list*] [**-lib_cell_type** *lib_cell_type*] [**-lib_model_name** *name* **-port_map {{***port_name net_ref***}***}]	
Arguments	*retention_name_list*	A list of target retention strategy names defined in *domain_name* using **set_retention** commands (see 6.49).
	-domain *domain_name*	The domain in which the strategies are defined.
	-elements *element_list*	A list of instances, named processes, or sequential `reg` or signal names whose respective sequential elements shall be mapped as specified.
	-exclude_elements *exclude_list*	A list of instances, named processes, or sequential `reg` or signal names whose respective sequential elements shall be excluded from mapping.
	-lib_cells *lib_cell_list*	A list of library cell names. Each cell in the list has retention behavior and is otherwise identical to the inferred RTL behavior of the underlying sequential element.
	-lib_cell_type *lib_cell_type*	The attribute of the library cells used to identify cells that have retention behavior and are otherwise identical to the inferred RTL behavior of the underlying sequential element.
	-lib_model_name *model_name* **-port_map {{***port_name net_ref***}***}	The name of the library cell or behavioral model and associated port connectivity.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **map_retention_cell** command constrains retention strategy implementation choices and may also specify functional retention behavior for verification.

-elements identifies elements from the *effective_element_list* (see 5.10) from a retention strategy in *retention_name_list*. If **-elements** is not specified, the *aggregate_element_list* for this command contains all elements from the *effective_element_list* of the *retention_name_list*.

It shall be an error if at least one of **-lib_cells**, **-lib_cell_type**, or **-lib_model_name** is not specified.

— If **-lib_cells** is specified, each cell shall be either defined by the **define_retention_cell** command (see 7.7) or defined in the Liberty file with required attributes;
If **-lib_cells** is specified, a retention cell from *lib_cell_list* shall be used; if **-lib_cell_type** is specified, a retention cell with the same type string specified by **define_retention_cell -cell_type** shall be used to implement the functionality specified by the corresponding retention strategy; if **-lib_cells** and **-lib_cell_type** are both specified, a retention cell from *lib_cell_list* that is also defined with the same type string in **define_retention_cell -cell_type** shall be used. Verification semantics are unchanged by the presence or absence of **-lib_cells** or **-lib_cell_type**.

— If **-lib_model_name** is specified, *model_name* shall be used as the verification model, and supply and logic ports shall be connected as specified by **-port_map** options; automatic corruption and retention verification semantics do not apply to a **-lib_model_name** model.

— If **-lib_model_name** is not specified, the verification semantic is that of the inferred RTL behavior of the underlying sequential element modified by the retention behavior prescribed by the applicable **set_retention** strategy.

Table 6 summarizes the semantics for combinations of **-lib_cells**, **-lib_cell_type**, and **-lib_model_name**.

Table 6—map_retention_cell option combinations

-lib_cells	-lib_cell_type	-lib_model_name	Verification semantic	Implementation cell constrained to
N	N	N	ERROR	ERROR
N	N	Y	*model_name*	*model_name*
N	Y	N	RTL with retention	*lib_cell_type*
N	Y	Y	*model_name*	*lib_cell_type*
Y	N	N	RTL with retention	*lib_cell_list*
Y	N	Y	*model_name*	*lib_cell_list*
Y	Y	N	RTL with retention	A cell from *lib_cell_list* that also has *lib_cell_type*
Y	Y	Y	*model_name*	A cell from *lib_cell_list* that also has *lib_cell_type*

For verification, an inferred register is assumed to have the following generic canonical interface:

— **CLOCK**—The signal whose rising edge triggers the register to load data.

— **DATA**—The signal whose value represents the next state of the register.

— **ASYNC_LOAD**—The signal that causes the register to load data when its value is one (`1`).

— **OUTPUT**—The signal that propagates the register output to the receivers of the register.

-port_map connects the specified *net_ref* to a *port* of the model. A *net_ref* may be one of the following:

a) A logic net name

b) A supply net name

c) One of the following symbolic references

1) **retention_ref**.*function_name*

This names a retention supply set function, where *function_name* refers to the supply net corresponding to the function it provides to the retention *ret_supply_set* (see 6.49).

2) **primary_ref**.*function_name*

This names a primary supply set function, where *function_name* refers to the supply net corresponding to the function it provides to the primary supply set of the domain.

3) save_signal

i) Refers to the save signal specified in the corresponding retention strategy.

ii) To invert the sense of the save signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *ret_supply_set* from the corresponding **set_retention** command (see 6.49).

4) restore_signal

i) Refers to the restore signal specified in the corresponding retention strategy.

ii) To invert the sense of the restore signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *ret_supply_set* from the corresponding **set_retention** command (see 6.49).

5) UPF_GENERIC_CLOCK

i) Refers to the canonical **CLOCK**.

ii) To invert the sense of the clock signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *primary_supply_set*.

6) **UPF_GENERIC_DATA**

i) Refers to the canonical **DATA**.

ii) To invert the sense of the data signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *primary_supply_set*.

7) UPF_GENERIC_ASYNC_LOAD

i) Refers to the canonical **ASYNC_LOAD**.

ii) To invert the sense of the asynchronous load signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *primary_supply_set*.

8) UPF_GENERIC_OUTPUT

i) Refers to the canonical **OUTPUT**.

ii) To invert the sense of the output signal, the Verilog bit-wise negation operator ~ can be specified before the *net_ref*. The logic inferred by the negation shall be implicitly connected to the *primary_supply_set*.

If **UPF_GENERIC_OUTPUT** is not explicitly mapped and the model has exactly one output port, that output port shall automatically be connected to the net that propagates the register output to the receivers of the register.

NOTE—All **map_*** commands specify the elements to be used rather than inferred through a strategy. The behavior of this manual mapping may lead to an implementation that is different from the RTL specification. Therefore, it may not be possible for logical equivalence checking tools to verify the equivalence of the mapped element to its RTL specification.

It shall be an error if

— *retention_name_list* is an empty list.

— *domain_name* does not indicate a previously created power domain.

— A retention strategy in *retention_name_list* does not indicate a previously defined retention strategy.

— An element in *element_list* is not included in the element list of a targeted retention strategy.
— Any retention strategy in *retention_name_list* does not specify signals needed to provide connection of the mapped functions.
— After completing the *port* and *net_ref* connections, any input port is unconnected, or no output port is connected to the net that propagates the register output to the receivers of the register.
— In implementation, none of the specified models in *lib_cell_list* implements the functionality specified by a targeted retention strategy.
— In implementation, none of the specified models having a *lib_cell_type* attribute implements the functionality specified by a targeted retention strategy.
— In implementation, none of the specified models in *lib_cell_list* that have a *lib_cell_type* attribute, when both are specified, implements the functionality specified by a targeted retention strategy.

Syntax example:

```
map_retention_cell {my_PDA_ret_strat_1 my_PDA_ret_strat_2 my_PDA_ret_strat_3}
    -domain PowerDomainA
    -elements {foo/U1 foo/U2}
   -lib_cells {RETFFIMP1 RETFFIMP2}
   -lib_cell_type FF_CKLO
    -lib_model_name RETFFVER -port_map {
       {CP       UPF_GENERIC_CLOCK}
       {D        UPF_GENERIC_DATA}
       {SET      UPF_GENERIC_ASYNC_LOAD}
       {SAVE     save_signal}
       {RESTORE  restore_signal}
       {VDDC     primary_supply_set.power}
       {VDDRET   ret_supply_set.power}
       {VSS      primary_supply_set.ground} }
```

6.34 merge_power_domains [deprecated]

This is a deprecated command; see also 6.1 and Annex D.

6.35 name_format

Purpose	Define the format for constructing names of implicitly created objects	
Syntax	**name_format** [**-isolation_prefix** *string*] [**-isolation_suffix** *string*] [**-level_shift_prefix** *string*] [**-level_shift_suffix** *string*] [**- implicit_supply_suffix** *string*] [**- implicit_logic_prefix** *string*] [**- implicit_logic_suffix** *string*]	
Arguments	**-isolation_prefix** *string*	The string prepended in front of an existing signal or port name to create a new name used during the introduction of a new isolation cell. The default value is the empty string "" or `NULL`.
	-isolation_suffix *string*	The string appended to the end of an existing signal or port name to create a new name used during the introduction of a new isolation cell. The default value is the string `_UPF_ISO`.
	-level_shift_prefix *string*	The string prepended in front of an existing signal or port name to create a new name used during the introduction of a new level-shifter cell. The default value is the empty string "" or `NULL`.
	-level_shift_suffix *string*	The string appended to the end of an existing signal or port name to create a new name used during the introduction of a new level-shifter cell. The default value is the string `_UPF_LS`.
	-implicit_supply_suffix *string*	The string appended to an existing supply net or port name to create a unique name for an implicitly created supply net or port. The default value is the string `_UPF_IS`.
	-implicit_logic_prefix *string*	The string prepended in front of an existing logic net or port name to create a unique name for an implicitly created logic net or port. The default value is `NULL`.
	-implicit_logic_suffix *string*	The string appended to an existing logic net or port name to create a unique name for an implicitly created logic net or port.The default value is the string `_UPF_IL`.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

Inferred objects have names in the logic design. The name for these objects is constructed as follows:

a) The base name of implicitly created objects is the name of the port or net being isolated or level-shifted, or the supply net, logic net, or port implicitly created to facilitate the connection of a net across hierarchy boundaries.

b) Any specified prefix is then prepended to the base name.

c) Any specified suffix is also appended to the base name.

d) If multiple prefixes or suffixes apply to the same object, they shall be added in the alphabetical order of the option name, e.g., **isolation_prefix** followed by **level_shift_prefix**.

If the generated name conflicts with another previously defined name in the same name space, the generated name is further extended by an underscore (_) followed by a positive integer. The value of the integer is the smallest number that makes the name unique in its name space. An empty string ("") is a valid value for any prefix or suffix option. When the prefix and suffix are both `NULL`, only the underscore (_) and number string combination are used as a suffix to disambiguate the name.

Different prefixes and suffixes may be specified in multiple calls to **name_format** (using different arguments). When **name_format** is specified with no options, the name format is reset to the default values shown in the *Arguments* list.

It shall be an error to specify an affix more than once.

Syntax example:

```
name_format -isolation_prefix "MY_ISO_" -isolation_suffix ""
```

A signal, `MY_ISO_FOO`, is created and connected to a new cell's output (to isolate the existing net `FOO`).

6.36 save_upf

Purpose	Create a UPF file of the structures relative to the active or specified scope	
Syntax	**save_upf** *upf_file_name* [**-scope** *instance_name*] [**-version** *string*]	
Arguments	*upf_file_name*	The UPF file to write.
	-scope *instance_name*	The scope relative to which the UPF commands are written.
Deprecated arguments	**-version** *string*	The UPF version of *upf_file_name*. See also 6.54. This is a deprecated option; see also 6.1 and Annex D.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **save_upf** command creates a UPF file that contains the power intent specified for a given scope. The power intent for that scope is written to file *upf_file_name*. The output file is generated after the power intent model has been constructed (see 8.3.2.)

If **-scope** *instance_name* is specified, the power intent is written for the specified scope. It is an error if this scope does not exist. Otherwise, the power intent is written for the current scope.

The following also apply:

a) Each invocation of **save_upf** generates a separate UPF output file.

b) If **save_upf** is invoked for two scopes and one is an ancestor of the other, then the file generated for the ancestor shall contain a duplicate of the information in the file generated for the other.

c) The following are equivalent:

```
save_upf <filename> -scope <instance>
```

and

```
set temp [set_scope <instance>]
save_upf <filename>
set_scope $temp
```

Syntax example:

```
save_upf test_suite1_Jan14
-scope top/proc_1
```

6.37 set_design_attributes

Purpose	Apply attributes to models or instances	
Syntax	**set_design_attributes** [**-models** *model_list*] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-attribute** {*name value*}]* [**-is_leaf_cell** [<**TRUE** | **FALSE**>]] [**-is_macro_cell** [<**TRUE** | **FALSE**>]]	
Arguments	**-models** *model_list*	A list of models to be attributed.
	-elements *element_list*	A list of rooted names: instances, named processes, sequential `regs`, or signal names.
	-exclude_elements *element_list*	A list of rooted names: instances, named processes, sequential `regs`, or signal names to exclude from the *effective_element_list* (see 5.10).
	-attribute {*name value*}	For the specified models or elements, associate the attribute *name* with the value of *value*. See Table 4.
	-is_leaf_cell [<**TRUE** | **FALSE**>]	If **-is_leaf_cell** is not specified at all, the default is **FALSE**. If **-is_leaf_cell** is specified without a value, the default value is **TRUE**. Equivalent to **-attribute {UPF_is_leaf_cell** *value*} (see 5.6).
	-is_macro_cell [<**TRUE** | **FALSE**>]	If **-is_macro_cell** is not specified at all, the default is **FALSE**. If **-is_macro_cell** is specified without a value, the default value is **TRUE**. Equivalent to **-attribute {UPF_is_macro_cell** *value*} (see 5.6).
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

This command sets the specified attributes for models or elements. It is an error if **set_design_attributes** is specified

a) with neither **-models** nor **-elements**; or

b) with both **-models** and **-elements**; or

c) with **-exclude_elements**, but not **-elements**; or

d) without at least one of **-attribute**, **-is_leaf_cell**, or **-is_macro_cell**.

A **UPF_is_leaf_cell** attribute value of **"TRUE"** on a model or instance prevents the **-transitive** processing for the descendants of the attributed model or instance for the following commands:

— **connect_supply_set** (see 6.12)

— **set_port_attributes** (see 6.46)

— **set_retention** (see 6.49)

— **set_retention_elements** (see 6.51)

— **find_objects** (see 6.26)

A **UPF_is_macro_cell** attribute value of **"TRUE"** on a model or instance causes any ports of an instance of the model to be recognized as part of the lower boundary of the power domain containing that instance if the driver or receiver supply of that port is specified as an attribute and is neither identical to nor equivalent to the primary supply of the containing power domain (see 6.17).

Examples

```
set_design_attributes -elements {lock_cache[0]}
   -attribute {UPF_is_leaf_cell TRUE}
set_design_attributes -models FIFO
   -attribute {UPF_is_leaf_cell TRUE}
set_design_attributes -models FIFO -is_leaf_cell
```

6.38 set_design_top

Purpose	Specify the design top module
Syntax	**set_design_top** *design_top_module*
Arguments	*design_top_module* The top module for which a UPF file was written.
Return value	Return an empty string.

The **set_design_top** command specifies the module for which this UPF file was written. See 4.2.7.

It is not an error if the instance to which this UPF file is applied is not an instance of the specified module. In particular, as long as the actual module has the same structure as the specified module, it may be possible to apply this UPF file to that module without errors. In this case, a tool may choose to issue a warning message.

Syntax example:

```
set_design_top ALU07
```

6.39 set_domain_supply_net [legacy]

Purpose	Set the default power and ground supply nets for a power domain	
Syntax	**set_domain_supply_net** *domain_name* **-primary_power_net** *supply_net_name* **-primary_ground_net** *supply_net_name*	
Arguments	*domain_name*	The domain where the default supply nets are to applied.
	-primary_power_net *supply_net_name*	The primary power supply net.
	-primary_ground_net *supply_net_name*	The primary ground net.
Return value	Return a `1` if successful or raise a `TCL_ERROR` if not.	

This is a legacy command; see also 6.1 and Annex D.

The **set_domain_supply_net** command associates the power and ground supply nets with the primary supply set for the domain.

The primary supply set's power and ground functions for the specified domain are associated with the corresponding power and ground supply net.

It shall be an error if

— *domain_name* does not indicate a previously created power domain.

— The primary supply set for *domain_name* already has a primary power or ground function association.

This command is semantically equivalent to

```
proc set_domain_supply_net {dn pp sn1 pg sn2} {
   if { string equal $pp "-primary_power_net" \
      && string equal $pg "-primary_ground_net"}{
      create_supply_set set_name -function {power $sn1}
         -function {ground $sn2}
      associate_supply_set set_name -handle $dn.primary
      return 1
   } else {
       return -code TCL_ERROR \
           -errorcode $ecode \
           -errorinfo $einfo \
           $resulttext
   } }
```

where any *italicized* arguments are implementation defined.

Syntax example:

```
set_domain_supply_net PD1
-primary_power_net PG1
-primary_ground_net PG0
```

6.40 set_equivalent

Purpose	Declare that supply nets or supply sets are electrically or functionally equivalent	
Syntax	**set_equivalent** [**-function_only**] [**-nets** *supply_net_name_list*] [**-sets** *supply_set_name_list*]	
Arguments	**-function_only**	Specifies that the supplies are functionally equivalent rather than electrically equivalent.
	-nets *supply_net_name_list*	A list of supply port and/or supply net names that are equivalent.
	-sets *supply_set_name_list*	A list of supply set names that are equivalent.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **set_equivalent** command declares that two or more supplies are *equivalent* (see 4.4.3).

If **–function_only** is specified, then the supplies are declared to be *functionally equivalent* only; otherwise the supplies are declared to be *electrically equivalent*, which implies that they are also functionally equivalent.

If **–nets** is specified, the command defines equivalence for a list of supply ports and/or supply nets. If **–sets** is specified, the command defines equivalence for a list of supply sets and/or supply set handles. One or the other of these options, but not both, shall be specified.

Equivalence of supply ports and nets can affect the number of sources for a given supply network and whether resolution is required (see 9.1). Equivalence of supply sets and supply set handles can affect various commands whose semantics are based on supply set identity or equivalence, including **create_composite_domain** (see 6.13), **create_power_domain** (see 6.17), **set_isolation** (see 6.41), **set_level_shifter** (see 6.43), **set_repeater** (see 6.48), and **set_port_attributes** (see 6.46).

For the input to an implementation tool, it shall be an error if electrical equivalence has been specified for two nets/sets but the actual connections that cause the electrical equivalence are not present in the UPF or in the HDL. For the input to a simulation tool, the actual connections implementing electrical equivalence need not be specified.

Syntax example:

```
set_equivalent -nets { vss1 vss2 ground }
set_equivalent -function_only -nets { vdd_wall vdd_battery }
set_equivalent -function_only -sets { /sys/aon_ss /mem/PD1.core_ssh }
```

6.41 set_isolation

<table>
<tr><td>Purpose</td><td colspan="3">Specify an isolation strategy</td></tr>
<tr><td>Syntax</td><td colspan="3">set_isolation <i>strategy_name</i>
    -domain <i>domain_name</i>
    [-elements <i>element_list</i>]
    [-exclude_elements <i>exclude_list</i>]
    [-source <<i>source_domain_name</i> | <i>source_supply_ref</i>>]
    [-sink <<i>sink_domain_name</i> | <i>sink_supply_ref</i>>]
    [-diff_supply_only [<TRUE | FALSE>]]
    [-use_equivalence [<TRUE | FALSE>]]
    [-applies_to <inputs | outputs | both>]
    [-applies_to_clamp <0 | 1 | any | Z | latch | <i>value</i>>]
    [-applies_to_sink_off_clamp <0 | 1 | any | Z | latch | <i>value</i>>]
    [-applies_to_source_off_clamp <0 | 1 | any | Z | latch | <i>value</i>>]
    [-no_isolation]
    [-force_isolation]
    [-location <self | other | parent | automatic | fanout | fanin | faninout | sibling>]
    [-clamp_value {< 0 | 1 | any | Z | latch | <i>value</i>>*}]
    [-isolation_signal <i>signal_list</i> [-isolation_sense {<high | low>*}]]
    [-isolation_supply_set <i>supply_set_list</i>]
    [-name_prefix <i>string</i>] [-name_suffix <i>string</i>]
    [-instance {{<i>instance_name port_name</i>}*}]
    [-update]
    [-sink_off_clamp {<0 | 1 | any | Z | latch | <i>value</i>> [<i>simstate_list</i>]}]
    [-source_off_clamp {<0 | 1 | any | Z | latch | <i>value</i>> [<i>simstate_list</i>]}]
    [-isolation_power_net <i>net_name</i>] [-isolation_ground_net <i>net_name</i>]
    [-transitive [<TRUE | FALSE>]]</td></tr>
<tr><td rowspan="8">Arguments</td><td>strategy_name</td><td colspan="2">The name of the isolation strategy.</td></tr>
<tr><td>-domain domain_name</td><td colspan="2">The domain for which this strategy is defined.</td></tr>
<tr><td>-elements element_list</td><td>A list of instances or ports to which the strategy potentially applies.</td><td>R</td></tr>
<tr><td>-exclude_elements exclude_list</td><td>A list of instances or ports to which the strategy does not apply.</td><td>R</td></tr>
<tr><td>-source <<i>source_domain_name</i> | <i>source_supply_ref</i>></td><td>The rooted name of a supply set or power domain. When a domain name is used, it represents the primary supply of that domain.</td><td>R</td></tr>
<tr><td>-sink <<i>sink_domain_name</i> | <i>sink_supply_ref</i>></td><td>The rooted name of a supply set or power domain. When a domain name is used, it represents the primary supply of that domain.</td><td>R</td></tr>
<tr><td>-diff_supply_only [<TRUE | FALSE>]</td><td>Indicates whether ports connected to other ports with the same supply should be isolated. The default is -diff_supply_only FALSE if the option is not specified at all; if -diff_supply_only is specified without a value, the default value is TRUE.</td><td>R</td></tr>
<tr><td>-use_equivalence [<TRUE | FALSE>]</td><td>Indicates whether to consider supply set equivalence. If –use_equivalence is not specified at all, the default is -use_equivalence TRUE; if –use_equivalence is specified without a value, the default value is TRUE.</td><td>R</td></tr>
<tr><td>-applies_to <inputs | outputs | both></td><td>A filter that restricts the strategy to apply only to ports of a given direction.</td><td>R</td></tr>
</table>

Arguments	**-applies_to_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>	A filter that restricts the strategy to apply only to ports with a particular clamp value requirement.	**R**
	-applies_to_sink_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	A filter that restricts the strategy to apply only to ports with a particular sink off clamp value requirement.	**R**
	-applies_to_source_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	A filter that restricts the strategy to apply only to ports with a particular source off clamp value requirement.	**R**
	-no_isolation	Specifies that isolation cells shall not be inserted on the specified ports.	**R**
	-force_isolation	Disables any implementation optimization involving isolation cells for a given strategy; used to force redundant isolation or to keep floating/constant ports that have an isolation strategy defined for them.	**R**
	-location <**self** | **other** | **parent** | **automatic** | **fanout** | **fanin** | **faninout** | **sibling**>	The location in which inferred isolation cells are placed in the logic hierarchy, which determines the power domain in which they will exist. The default is **self**.	**R**
	-clamp_value {<**0** | **1** | **any** | **Z** | **latch** | *value*>***}**	The value(s) that the isolation cell can drive. The default is **0**.	**R**
	-isolation_signal *signal_list*	The control signal(s) that cause(s) the isolation cell to drive with the corresponding value(s) specified in **–clamp_value**.	**R**
	-isolation_sense {<**high** | **low**>***}**	The active level(s) of the corresponding isolation signal(s) specified in **–isolation_signal**. The default is **high**.	**R**
	-isolation_supply_set *supply_set_list*	The supply set(s) that power the isolation cell for the corresponding control signal(s) specified in **–isolation_signal**.	**R**
	-name_prefix *string* **-name_suffix** *string*	The name format (prefix and suffix) for generated isolation instances or nets related to implementation of the isolation strategy.	**R**
	-instance {{*instance_name port_name*}***}**	The name of a technology leaf-cell instance and the name of the logic port that it isolates.	**R**
	-update	Indicates that this command provides additional information for a previous command with the same *strategy_name* and *domain_name* and executed in the same scope.	**R**
Legacy arguments	**-isolation_power_net** *net_name*	This option specifies the supply net used as the power for the isolation logic inferred by this strategy. This is a legacy option; see also 6.1 and Annex D.	**R**
	-isolation_ground_net *net_name*	This option specifies the supply net used as the ground for the isolation logic inferred by this strategy. This is a legacy option; see also 6.1 and Annex D.	**R**

Deprecated arguments	**-location fanin | faninout | sibling**	The isolation cell is placed at all fanin locations (sources) of the port being isolated. The isolation cell is placed at all fanout locations (sinks) for each output port being isolated, or at all fanin locations (sources) for each input port being isolated. A new sibling is created into which the isolation cells are placed in the logic hierarchy. These are all deprecated options; see also 6.1 and Annex D.	**R**
	-clamp_value {<any>}	The **-clamp_value** option **any**. This is a deprecated option; see also 6.1 and Annex D.	**R**
	-sink_off_clamp {<0 | 1 | any | Z | latch | *value*> [*simstate_list*]}	The **-sink_off_clamp** option specifies the clamp requirement when the sink domain is off. This is a deprecated option; see also 6.1 and Annex D.	**R**
	-source_off_clamp {<0 | 1 | any | Z | latch | *value*> [*simstate_list*]}	The **-source_off_clamp** option specifies the clamp requirement when the source domain is off. This is a deprecated option; see also 6.1 and Annex D.	**R**
	-transitive [<TRUE | FALSE>]	When **-transitive** is **TRUE** (the default), the command applies to the descendants of the elements. This is a deprecated option; see also 6.1 and Annex D.	
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **set_isolation** command defines an isolation strategy for ports on the interface of a power domain (see 6.17). An isolation strategy is applied at the domain boundary, as required to ensure correct electrical and logical functionality when domains are in different power states.

-domain specifies the domain for which this strategy is defined.

-elements explicitly identifies a set of candidate ports to which this strategy potentially applies. The *element_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *element_list*, it is equivalent to specifying all the ports of the instance in the *element_list*, but with lower precedence (see 5.8). Any *element_list*s specified on the base command or any updates (see **-update**) of the base command are combined. If **-elements** is not specified in the base command or any update, every port on the interface of the domain is included in the *aggregate_element_list* (see 5.10).

-exclude_elements explicitly identifies a set of ports to which this strategy does not apply. The *exclude_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *exclude_list*, it is equivalent to specifying all the ports of the instance in the *exclude_list*. Any *exclude_list*s specified on the base command or any updates of the base command are combined into the *aggregate_exclude_list* (see 5.10).

The arguments **-source**, **-sink**, **-diff_supply_only**, **-applies_to**, **-applies_to_clamp**, **-applies_to_sink_off_clamp**, and **-applies_to_source_off_clamp** serve as filters that further restrict the set of ports to which a given **set_isolation** command applies. The command only applies to those ports that satisfy all of the specified filters.

-source is satisfied by any port that is driven by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

-sink is satisfied by any port that is received by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

NOTE 1—A port that does not have a driver will never satisfy the **-source** filter. A port that does not have a receiver will never satisfy the **-sink** filter.

-diff_supply_only TRUE is satisfied by any port for which the driving logic and receiving logic are powered by supply sets that do not match (see **-use_equivalence**), or for which either driving or receiving or both supply sets cannot be determined. **-diff_supply_only FALSE** is satisfied by any port.

-use_equivalence specifies whether supply set equivalence is to be considered in determining when two supply sets match. If **-use_equivalence** is specified with the value *False*, the **-source** and **-sink** filters shall match only the named supply set; the **-diff_supply_only TRUE** filter shall be satisfied only if the driver supply and receiver supply of the port are not identical. Otherwise, the **-source** and **-sink** filters shall match the named supply set or any supply set that is equivalent to the named supply set; the **-diff_supply_only TRUE** filter shall be satisfied only if the driver supply and receiver supply of the port are neither identical nor equivalent.

-applies_to is satisfied by any port that has the specified mode. For upper boundary ports, this filter is satisfied when the direction of the port matches. For lower boundary ports, this filter is satisfied when the inverse of the direction of the port matches. For example, a lower boundary port with a direction `OUT` would satisfy the **-applies_to IN** filter, because an output from a lower boundary port is an input to this domain. **-applies_to** is always relative to the specified domain.

-applies_to_clamp, **-applies_to_sink_off_clamp**, and **-applies_to_source_off_clamp** are satisfied by any port that has the specified value for the **UPF_clamp_value**, **UPF_sink_off_clamp**, or **UPF_source_off_clamp** port attribute, respectively.

The *effective_element_list* (see 5.10) for this command consists of all the port names in the *aggregate_element_list* that are not also in the *aggregate_exclude_list* and that satisfy all of the filters specified in the command. If a port in the *effective_element_list* is not on the interface of the specified domain, it shall not be isolated.

If a given port name is referenced in the *effective_element_list* of more than one isolation strategy of a given domain, the precedence rules (see 5.8) determine which of those strategies actually apply to that port name. If the precedence rules identify multiple strategies that apply to the same port name, then those strategies shall each have a **-sink** filter that matches the receiving supply of a different sink domain for the specified port. It shall be an error if the precedence rules identify multiple strategies that apply to the same port name such that more than one strategy applies to the same sink domain for that port.

If **-no_isolation** is specified, then isolation is not inferred for any port in the *effective_element_list*.

If **-force_isolation** is specified, then isolation is inferred for each port in the *effective_element_list* and the inferred isolation cells are not to be optimized away, even if such optimization does not change the behavior of the design.

If neither **-no_isolation** nor **-force_isolation** is specified, then isolation is inferred for each port in the *effective_element_list*, and implementation tools are free to optimize away isolation cells that are redundant provided that such optimization does not change the behavior of the design.

-location defines where the isolation cells are placed in the logic hierarchy and therefore the power domain into which they are inserted, as follows:

self—the isolation cell is placed inside the domain whose interface port is being isolated (the default).

other—the isolation cell is placed in the parent for ports on the interface of the domain that connect to the parent, and in the child for ports on the interface of the domain that connect to a child.

parent—the isolation cell is placed in the parent of the instance whose interface port is being isolated.

fanout—the isolation cell is placed at all fanout locations (receiving logic) of the port being isolated.

automatic—the implementation tool is free to choose any of the locations **self**, **parent**, or **other**.

If **-location fanout** is specified, the isolation cell shall be inserted at the port on the domain boundary that is closest to the receiving logic. If the receiving logic is in a macro cell instance, the isolation cell shall be inserted in the domain that contains the macro cell instance; otherwise the isolation cell shall be inserted in the domain that contains the receiving logic.

If **-location automatic** is specified, and a second isolation strategy is also applied to this port by the other power domain sharing this interface, the location chosen by the tool shall be such that the isolation cell contributed by the source domain is placed closer to the driving logic and the isolation cell contributed by the sink domain is placed closer to the receiving logic.

If any pair of isolation cells are inferred from two different isolation strategies for ports of two different power domains along the same path from a driver to a receiver, and the **–location** specified results in both cells being inserted into the same domain, then the two isolation cells shall be inserted such that the isolation cell contributed by the source domain is placed closer to the driving logic and the isolation cell contributed by the sink domain is placed closer to the receiving logic.

If two or more isolation strategies apply to the same port on the interface of a power domain, such that multiple isolation cells need to be inferred for different paths from that port to a receiving domain, the **-location** specified explicitly or implicitly for each of those strategies shall be such that the various isolation cells can be inserted without splitting the port into multiple ports. It shall be an error if multiple isolation strategies for the same port cannot be implemented without duplicating the port.

The **-clamp_value**, **-isolation_signal** and **-isolation_sense**, and **-isolation_supply_set** options are each specified as a single value or a list. If any of these options specify a list, then all lists specified for these options shall be of the same length and any single value specified is treated as a list of values of the same length. The tuples formed by associating the positional entries from each list shall be used to define separate isolation requirements for the strategy. These tuples are applied to the isolation cell from the isolation cell's data input port to its data output port in the order in which they appear in each list. The output of the isolation cell shall be the right-most value in the **-clamp_value** list whose corresponding isolation signal is active.

-clamp_value specifies the value of the inferred isolation cell's output when isolation is enabled. The specification may be a single value or a list of values. Any of the following may be specified:

0 (the logic value `0`)

1 (the logic value `1`)

Z (the logic value `Z`)

latch (the value of the non-isolated port when the isolation signal becomes active)

value specifies a value that is legal for the type of the port, e.g., `255` might be specified for an integer-typed port (perhaps constrained to an unsigned 8-bit range).

If **-clamp_value** is not specified, it defaults to `0`.

Verification shall issue an error when a **UPF_sink_off_clamp**, **UPF_source_off_clamp**, or **UPF_clamp_value** requirement is violated.

-isolation_signal identifies the control signal for each clamp value specified by **-clamp_value**.

-isolation_sense specifies the value that enables isolation, for each signal specified by **-isolation_signal**.

-isolation_supply_set specifies the supply set(s) that shall be used to power the inferred isolation cell. The isolation supply set(s) specified by **-isolation_supply_set** are implicitly connected to the isolation logic inferred by this command. If **-isolation_supply_set** is not specified, the **default_isolation** supply of the power domain in which the inferred cell will be located is used as the isolation supply. For example, if **set_isolation** is specified with **-location parent** and **-isolation_supply_set** is not specified, then the **default_isolation** supply set of the parent domain is used.

-name_prefix specifies the substring to place at the beginning of any generated name implementing this strategy.

-name_suffix specifies the substring to place at the end of any generated name implementing this strategy.

-instance specifies that the isolation functionality exists in the HDL design and *instance_name* denotes the instance providing part or all of this functionality. An *instance_name* is a simple name or hierarchical name rooted in the current scope. If an empty string appears as an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance should not be reinferred or reimplemented by subsequent tool runs.

In this case, the following also apply:

- Isolation enable signal(s) are automatically connected to one or more ports of an instance of a cell defined by the library command **define_isolation_cell** (see 7.4). If the strategy specifies multiple isolation enable signals, then the cell shall also be defined with both the **-enable** option and the **-aux_enables** option (see 7.4), the first isolation enable signal shall be connected to the port specified by the **-enable** option, and the rest of the signals shall be connected to the ports specified by the **-aux_enables** option in the same order.
- If the strategy specifies a single isolation supply set, the supply nets of the set shall be automatically connected to the primary supply ports of the isolation cell. If the strategy specifies multiple isolation supply sets, the isolation enable ports shall have related power, ground, and bias port attributes (see 6.45 and 6.46), and the supply nets of the isolation supply set corresponding to each isolation enable signal shall be automatically connected to the supply ports matching the related power, ground, and bias ports of the isolation enable port (see 7.4).
- If there are no supply ports on the instance, then the isolation supply set(s) specified in the strategy shall be implicitly connected to the instance.
- It is an error if there is a single isolation enable signal and there is more than one port on the library cell of the instance defined as isolation enable pin or aux enable pin (see 7.4).

–update adds information to the base command executed in the same scope. When specified with **-update**, **-elements** and **-exclude_elements** are additive: the set of instances or ports in the *aggregate_elements_list* is the union of all **-elements** specifications given in the base command and any update of this command, and the *aggregate_exclude_list* is the union of all **-exclude_elements** specifications given in the base command and any update of this command.

Tools shall not use information about system power states to avoid inserting isolation as directed by these strategies. However, tools may optionally use information about system power states to issue a warning that certain strategies appear to be unnecessary.

The following also apply:

— This command never applies to inout ports.

— It is erroneous if an isolation strategy isolates its own control signal.

— It shall be an error if **-no_isolation** is specified along with any of the following: **-force_isolation**, **-isolation_signal**, **-isolation_sense**, **-instance**, **-location**, **-name_prefix**, **-name_suffix**, **-isolation_supply_set**, **-isolation_power_net**, or **-isolation_ground_net**.

— It shall be an error if the isolation supply set is not defined for a strategy and the domain in which the inferred isolation cell is located does not have a **default_isolation** supply set.

NOTE 2—To specify an isolation strategy for a port `P` on the lower boundary of a power domain `D` (see 4.3.1), a **set_isolation** command can specify `-domain D` and specify the port name `I/P`, where `I` is the hierarchical name of an instance that is instantiated in domain `D` but is not in the extent of domain `D`, and `P` is the simple name of the port of that instance. The combination of the **-domain** specification and the hierarchical port name makes it clear this reference is to the HighConn of the specified port, which is part of the lower boundary of the domain `D`.

NOTE 3—The *exclude_list* in **-exclude_elements** can specify instances or ports that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

NOTE 4—If a **-diff_supply_only**, **-source**, or **-sink** argument is used and instances are included in designs with different power distribution or connectivity, the evaluation of the need for isolation may vary and cause a change in the logical function of a block.

NOTE 5—Isolation clamp value port properties can be annotated in HDL using the attributes shown in 5.6. The same attributes may be specified using the **set_port_attributes** command in 6.46.

NOTE 6—It is not an error if multiple isolation strategies apply to a connection from one domain to another domain.

Syntax example:

```
set_isolation parent_strategy
  -domain pda
  -elements {a b c d}
  -isolation_supply_set {pda_isolation_supply}
   -clamp_value {1}
-applies_to outputs -sink pdb

set_isolation parent_strategy -update
   -domain pda
   -isolation_signal cpu_iso
   -isolation_sense low -location parent
```

6.42 set_isolation_control [deprecated]

This is a deprecated command; see also 6.1 and Annex D.

6.43 set_level_shifter

Purpose	Specify a level-shifter strategy		
Syntax	**set_level_shifter** *strategy_name* **-domain** *domain_name* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-source** <*source_domain_name* | *source_supply_ref*>] [**-sink** <*sink_domain_name* | *sink_supply_ref*>] [**-use_equivalence** [<**TRUE** | **FALSE**>]] [**-applies_to** <**inputs** | **outputs** | **both**>] [**-rule** <**low_to_high** | **high_to_low** | **both**>] [**-threshold** <*value* | *list*>] [**-no_shift**] [**-force_shift**] [**-location** <**self** | **other** | **parent** | **automatic** | **fanout** | **fanin** | **faninout** | **sibling**>] [**-input_supply_set** *supply_set_ref*] [**-output_supply_set** *supply_set_ref*] [**-internal_supply_set** *supply_set_ref*] [**-name_prefix** *string*] [**-name_suffix** *string*] [**-instance** {{*instance_name port_name*}*}] [**-update**] [**-transitive** [<**TRUE** | **FALSE**>]]		
Arguments	*strategy_name*	The name of the level-shifter strategy.	
	-domain *domain_name*	The domain for which this strategy is defined.	
	-elements *element_list*	A list of instances or ports to which the strategy potentially applies.	**R**
	-exclude_elements *exclude_list*	A list of instances or ports to which the strategy does not apply.	**R**
	-source <*source_domain_name* | *source_supply_ref*>	The rooted name of a supply set or power domain. When a domain name is used, it represents the primary supply of that domain.	**R**
	-sink <*sink_domain_name* | *sink_supply_ref*>	The rooted name of a supply set or power domain. When a domain name is used, it represents the primary supply of that domain.	**R**
	-use_equivalence [<**TRUE** | **FALSE**>]	Indicates whether to consider supply set equivalence. If **-use_equivalence** is not specified at all, the default is **-use_equivalence TRUE**; if **–use_equivalence** is specified without a value, the default value is **TRUE**.	**R**
	-applies_to <**inputs** | **outputs** | **both**>	A filter that restricts the strategy to apply only to ports of a given direction.	**R**
	-rule <**low_to_high** | **high_to_low** | **both**>	A filter that restricts the strategy to apply only to ports that require a given level-shifting direction. The default is **both**.	**R**
	-threshold <*value* | *list*>	A filter that restricts the strategy to apply only to ports that involve a voltage difference above a certain threshold. The default is `0`.	**R**
	-no_shift	Specifies that level-shifter cells shall not be inserted on the specified ports.	**R**
	-force_shift	Disables any implementation optimization involving level-shifter cells for a given strategy.	**R**

Arguments	**-location** <**self** | **other** | **parent** | **automatic** | **fanout** | **fanin** | **faninout** | **sibling**>	The location in which inferred level-shifter cells are placed in the logic hierarchy, which determines the power domain in which they will exist. The default is **self**.	**R**
	-input_supply_set *supply_set_ref*	The supply set used to power the input portion of the level-shifter.	**R**
	-output_supply_set *supply_set_ref*	The supply set used to power the output portion of the level-shifter.	**R**
	-internal_supply_set *supply_set_ref*	The supply set used to power internal circuits within the level-shifter.	**R**
	-name_prefix *string* **-name_suffix** *string*	The name format (prefix and suffix) for generated level-shifter instances or nets related to implementation of the level-shifting strategy.	**R**
	-instance {{*instance_name port_name*}*}	The name of a technology library leaf-cell instance and the name of the logic port that it level-shifts.	**R**
	-update	Indicates that this command provides additional information for a previous command with the same *strategy_name* and *domain_name* and executed in the same scope.	**R**
Deprecated arguments	**-threshold** <*list*>	*list* is a matrix of values. This is a deprecated option; see also 6.1 and Annex D.	**R**
	-location fanin | **faninout** | **sibling**	The level-shifter cell is placed at all fanin locations (sources) of the port being shifted. The level-shifter cell is placed at all fanout locations (sinks) for each output port being shifted, or at all fanin locations (sources) for each input port being shifted. A new sibling is created into which the level-shifter cells are placed in the logic hierarchy. These are all deprecated options; see also 6.1 and Annex D.	**R**
	-transitive [<**TRUE** | **FALSE**>]	When **-transitive** is **TRUE** (the default), the command applies to the descendants of the elements. This is a deprecated option; see also 6.1 and Annex D.	
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **set_level_shifter** command defines a level-shifting strategy for ports on the interface of a power domain (see 6.17). A level-shifter strategy is applied at the domain boundary, as required to correct for voltage differences between driving and receiving supplies of a port.

-domain specifies the domain for which this strategy is defined.

-elements explicitly identifies a set of candidate ports to which this strategy potentially applies. The *element_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *element_list*, it is equivalent to specifying all the ports of the instance in the *element_list* but with lower precedence (see 5.8). Any *element_list*s specified on the base command or any updates (see **-update**) of the base command are combined. If **-elements** is not specified in the base command or any update, every port on the interface of the domain is included in the *aggregate_element_list* (see 5.10).

-exclude_elements explicitly identifies a set of ports to which this strategy does not apply. The *exclude_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *exclude_list*, it is equivalent to specifying all the ports of the instance in the *exclude_list*. Any *exclude_list*s specified on the base command or any updates of the base command are combined into the *aggregate_exclude_list* (see 5.10).

The arguments **-source**, **-sink**, **-applies_to**, **-rule**, and **-threshold** serve as filters that further restrict the set of ports to which a given **set_level_shifter** command applies. The command only applies to those ports that satisfy all of the specified filters.

-source is satisfied by any port that is driven by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

-sink is satisfied by any port that is received by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

NOTE 1—A port that does not have a driver will never satisfy the **-source** filter. A port that does not have a receiver will never satisfy the **-sink** filter.

-use_equivalence specifies whether supply set equivalence is to be considered in determining when two supply sets match. If **-use_equivalence** is specified with the value *False*, the **-source** and **-sink** filters shall match only the named supply set. Otherwise, the **-source** and **-sink** filters shall match the named supply set or any supply set that is equivalent to the named supply set.

-applies_to is satisfied by any port that has the specified mode. For upper boundary ports, this filter is satisfied when the direction of the port matches. For lower boundary ports, this filter is satisfied when the inverse of the direction of the port matches. For example, a lower boundary port with a direction `OUT` would satisfy the **-applies_to IN** filter, because an output from a lower boundary port is an input to this domain. **-applies_to** is always relative to the specified domain.

-rule is satisfied by any port for which the driving and receiving logic have the specified voltage relationship. If **low_to_high** is specified, a given port satisfies this filter if the voltage of its driver supply is less than the voltage of its receiver supply. If **high_to_low** is specified, a given port satisfies this filter if the voltage of its driver supply is greater than the voltage of its receiver supply. If **-rule both** is specified, a given port satisfies this filter if would satisfy either **-rule low_to_high** or **-rule high_to_low**.

-threshold is satisfied by any port for which the magnitude of the difference between the driver and receiver supply voltages can exceed a specified threshold value. The nominal power and ground of the port's driver supply are compared with the nominal power and ground of the port's receiver supply. This option requires tools to use information defined in power states of the supplies involved in a given interconnection between objects with different supplies. If **-threshold** is not specified, it defaults to `0`, which ensures that a level-shifter will be inserted for a given port if there is any voltage difference.

The **-threshold** *value* is evaluated as shown in the following pseudo code:

```
foreach A in the legal power states of the input supply set
   foreach B in the legal power states of the output supply set
      if exists legal power state (A, B)
         if (threshold value < max (|(A_nominal_power - B_nominal_power)|,
               |(A_nominal_ground - B_nominal_ground)|))
            return(REQUIRED)
         endif
      endif
```

```
    next B
  next A
  return (NOT REQUIRED)
```

The *effective_element_list* (see 5.10) for this command consists of all the port names in the *aggregate_element_list* that are not also in the *aggregate_exclude_list* and that satisfy all of the filters specified in the command. If a port in the *effective_element_list* is not on the interface of the specified domain, it shall not be level-shifted.

If a given port name is referenced in the *effective_element_list* of more than one level-shifting strategy of a given domain, the precedence rules (see 5.8) determine which of those strategies actually apply to that port name. If the precedence rules identify multiple strategies that apply to the same port name, then those strategies shall each have a **-sink** filter that matches the receiving supply of a different sink domain for the specified port. It shall be an error if the precedence rules identify multiple strategies that apply to the same port name such that more than one strategy applies to the same sink domain for that port.

If **-no_shift** is specified, then level-shifting is not inferred for any port in the *effective_element_list*.

If **-force_shift** is specified, then level-shifting is inferred for each port in the *effective_element_list* and the inferred level-shifting cells are not to be optimized away, even if such optimization does not change the behavior of the design.

If neither **-no_shift** nor **-force_shift** is specified, then level-shifting is inferred for each port in the *effective_element_list*, and implementation tools are free to optimize away level-shifting cells that are redundant provided that such optimization does not change the behavior of the design.

-location defines where the level-shifter cells are placed in the logic hierarchy and therefore the power domain into which they are inserted, as follows:

self—the level-shifter cell is placed inside the domain whose interface port is being shifted (the default).

other—the level-shifter cell is placed in the parent for ports on the interface of the domain that connect to the parent, and in the child for ports on the interface of the domain that connect to a child.

parent—the level-shifter cell is placed in the parent of the element whose interface port is being shifted.

fanout—the level-shifter cell is placed at all fanout locations (receiving logic) of the port being shifted.

automatic—the implementation tool is free to choose any of the locations **self**, **parent**, or **other**.

If **-location fanout** is specified, the level-shifter cell shall be inserted at the port on the domain boundary that is closest to the receiving logic.

If the port at which the level-shifter is inserted is connected to the input or output of an isolation cell, or is connected to the output of one isolation cell and the input of another isolation cell, the level-shifter is inserted either immediately before, or immediately after, or between the isolation cell(s), as appropriate, to achieve the best match between any explicitly specified input/output supplies of the strategy and the actual driver/receiver supplies at each location.

If multiple level-shifter strategies are defined that would insert a level-shifter at the same domain boundary, any of those level-shifter strategies can be applied in any of the preceding locations, in either domain, either singly or in combination. If two potential solutions match the driving and receiving supplies equally well, the solution that applies a level-shifting strategy contributed by a domain closer to the receiving domain shall be used.

For a port on the boundary between two domains, if neither domain explicitly defines a level-shifter strategy that applies to that port, then a default level-shifter strategy is implicitly defined for the LowConn side of the port, on the upper boundary of the lower domain. The default level-shifter strategy is as follows:

```
set_level_shifter -domain <domain name> -elements <port name> -rule both
   -threshold 0
```

-input_supply_set specifies the supply set connected to input supply ports of the level-shifter (see 7.4). The default is the supply of the logic driving the level-shifter input. The default is used if and only if that supply set is available in the domain in which the level-shifter will be located. It shall be an error if the default supply set is required but is not available.

-output_supply_set specifies the supply set connected to the output supply ports of the level-shifter (see 7.4). The default is the supply of the logic receiving the level-shifter output. The default is used if and only if that supply set is available in the domain in which the level-shifter will be located. It shall be an error if the default supply set is required but is not available.

Default input and output supply set definitions apply only if exactly one level-shifter strategy applies to a given port, all drivers of that port have equivalent supplies, and all receivers of that port have equivalent supplies. For more complex cases, the required supply sets should be explicitly specified.

If the level-shifter strategy is mapped to a library cell that requires only a single supply, then explicit specification of an input supply set is not required, any explicit input supply set specification is ignored, and the default input supply set does not apply; only the output supply set is used.

-internal_supply_set specifies the supply set that shall be used to provide power to supply ports that are not related to the inputs or outputs of the level-shifter. There is no default supply set defined for **-internal_supply_set.**

-name_prefix specifies the substring to place at the beginning of any generated name implementing this strategy.

-name_suffix specifies the substring to place at the end of any generated name implementing this strategy.

-instance specifies that the level-shifter functionality exists in the HDL design, and *instance_name* denotes the instance providing part or all of this functionality. An *instance_name* is a simple name or hierarchical name rooted in the current scope. If an empty string appears as an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance should not be reinferred or reimplemented by subsequent tool runs.

–update adds information to the base command executed in the same scope. When specified with **-update**, **-elements** and **-exclude_elements** are additive: the set of instances or ports in the *aggregate_elements_list* is the union of all **-elements** specifications given in the base command and any update of this command, and the *aggregate_exclude_list* is the union of all **-exclude_elements** specifications given in the base command and any update of this command.

The following also apply:

- This command never applies to inout ports.
- The simstate semantics of all implicitly connected supply sets apply to the output of a level-shifter.
- It shall be an error if **-no_shift** is specified along with any of the following: **-force_shift**, **-instance**, **-location**, **-name_prefix**, **-name_suffix**, **-input_supply_set**, **-output_supply_set**, or **-internal_supply_set**.

It shall be an error if there is a connection between a driver and receiver and all of the following apply:

— The supplies powering the driver and receiver are at different voltage levels.

— A level-shifter is not specified for the connection using a level-shifter strategy.

— A level-shifter cannot be inferred for the connection by analysis of the power states of the supplies to the driver and receiver.

NOTE 2—To specify a level-shifting strategy for a port `P` on the lower boundary of a power domain `D`, a **set_level_shifter** command can specify `-domain D` and specify the port name `I/P`, where `I` is the hierarchical name of an instance that is instantiated in domain `D` but is not in the extent of domain `D`, and `P` is the simple name of the port of that instance. The combination of the **-domain** specification and the hierarchical port name makes it clear this reference is to the HighConn of the specified port, which is part of the lower boundary of the domain `D`.

NOTE 3—The *exclude_list* in **-exclude_elements** can specify instances or ports that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

NOTE 4—It is not an error if multiple level-shifting strategies apply to a connection from one domain to another domain.

Syntax example:

```
set_level_shifter shift_up
  -domain PowerDomainZ
  -applies_to inputs -source PowerDomainX.ss1
  -threshold 0.02
  -rule both
set_level_shifter TurnOffDefaultLS -domain PD -no_shift
//this turns off inference of a default level-shifter for ports on the
//upper boundary of domain PD
```

6.44 set_partial_on_translation

Purpose	Define the translation of **PARTIAL_ON**
Syntax	**set_partial_on_translation** <**OFF** | **FULL_ON**>
Arguments	**OFF** | **FULL_ON** The value to use in place of **PARTIAL_ON**.
Return value	Return the setting of the translation if successful or raise a `TCL_ERROR` if not.

This command defines the translation of **PARTIAL_ON** to **FULL_ON** or **OFF** for purposes of evaluating the power state of supply sets and power domains. The state of a supply *set* is evaluated after **PARTIAL_ON** is translated to **FULL_ON** or **OFF** for each supply *net* in the set.

It shall be an error if this command is invoked with different values in the same UPF description.

Syntax example:

```
set_partial_on_translation FULL_ON
```

6.45 set_pin_related_supply [deprecated]

This is a deprecated command; see also 6.1 and Annex D.

6.46 set_port_attributes

Purpose	Define information on ports
Syntax	**set_port_attributes** [**-model** *name*] [**-elements** *element_list*] [**-exclude_elements** *element_exclude_list*] [**-ports** *port_list*] [**-exclude_ports** *port_exclude_list*] [**-applies_to** <**inputs** | **outputs** | **both**>] [**-attribute** {*name value*}]* [**-clamp_value** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-sink_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-source_off_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-driver_supply** *supply_set_ref*] [**-receiver_supply** *supply_set_ref*] [**-pg_type** *pg_type_value*] [**-related_power_port** *supply_port_name*] [**-related_ground_port** *supply_port_name*] [**-related_bias_ports** *supply_port_name_list*] [**-feedthrough**] [**-unconnected**] [{**-domains** *domain_list* [**-applies_to** <**inputs** | **outputs** | **both**>]}] [{**-exclude_domains** *domain_list* [**-applies_to** <**inputs** | **outputs** | **both**>]}] [**-repeater_supply** *supply_set_ref*] [**-transitive** [<**TRUE** | **FALSE**>]]

Arguments

-model *name*	A module or library cell whose ports are to be attributed.
-elements *element_list*	A list of instances whose ports are to be attributed.
-exclude_elements *element_exclude_list*	A list of instances whose ports are to be excluded from the command.
-ports *port_list*	A list of simple names of ports to be attributed.
-exclude_ports *port_exclude_list*	A list of ports to be excluded from the command.
-applies_to <**inputs** | **outputs** | **both**>	Indicates whether the specified input ports, output ports, or both are to be attributed.
-attribute {*name value*}	The attribute *name* and *value* pair to be associated with the specified ports.
-clamp_value <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement. Equivalent to **-attribute {UPF_clamp_value** *value***}** (see 5.6).
-sink_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement when the sink domain is off. Equivalent to **-attribute {UPF_sink_off_clamp_value** *value***}** (see 5.6).
-source_off_clamp <**0** | **1** | **any** | **Z** | **latch** | *value*>	The clamp requirement when the source domain is off. Equivalent to **-attribute {UPF_source_off_clamp_value** *value***}** (see 5.6).
-driver_supply *supply_set_ref*	The supply set used by drivers of the port. Equivalent to **-attribute {UPF_driver_supply** *supply_set_ref***}** (see 5.6).
-receiver_supply *supply_set_ref*	The supply set used by receivers of the port. Equivalent to **-attribute {UPF_receiver_supply** *supply_set_ref***}** (see 5.6).
-pg_type *pg_type_value*	The *pg_type* port. Equivalent to **-attribute {UPF_pg_type** *pg_type_value***}** (see 5.6).

Arguments	**-related_power_port** *supply_port_name*	The power port for the attributed port. Equivalent to **-attribute {UPF_related_power_port** *supply_port_name***}** (see 5.6).
	-related_ground_port *supply_port_name*	The ground port for the attributed port. Equivalent to **-attribute {UPF_related_ground_port** *supply_port_name***}** (see 5.6).
	-related_bias_ports *supply_port_name_list*	The bias port(s) for the attributed port. Equivalent to **-attribute {UPF_related_bias_ports** *supply_port_name_list***}** (see 5.6).
	-feedthrough	Indicates that the specified ports are connected together internally to form a feedthrough. Equivalent to **-attribute {UPF_feedthrough TRUE}** (see 5.6).
	-unconnected	Indicates that the specified ports are not connected at all internally. Equivalent to **-attribute {UPF_unconnected TRUE}** (see 5.6).
Deprecated arguments	**{-domains** *domain_list* **[-applies_to <inputs | outputs | both>]}**	A list of domains whose ports are to be attributed.This is a deprecated option; see also 6.1 and Annex D.
	{-exclude_domains *domain_list* **[-applies_to <inputs | outputs | both>]}**	A list of domains whose ports are excluded from being attributed. This is a deprecated option; see also 6.1 and Annex D.
	-repeater_supply *supply_set_ref*	The supply set used by a repeater driving the port. This is a deprecated option; see also 6.1 and Annex D.
	-transitive [<TRUE | FALSE>]	If **-transitive** is not specified at all, the default is **-transitive TRUE**. If **-transitive** is specified without a value, the default value is **TRUE**. This is a deprecated option; see also 6.1 and Annex D.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **set_port_attributes** command specifies information associated with ports of models or instances. Certain predefined attributes identify a port's related supplies and in doing so may define the lower boundary of a power domain; other predefined attributes provide information relevant to isolation and level-shifting insertion.

User-defined attributes may also be associated with a port. The meaning of a user-defined attribute is not specified by this standard.

The set of ports attributed is determined as follows:

a) A set of candidate ports is first identified. This set includes the following:

   1) If **-elements** is specified, all ports of each instance named in the elements list are included in the candidate set, including any logic ports inferred from **create_logic_port** (see 6.16), but excluding any supply ports.

   2) If **-ports** is specified, each port named in the ports list is included in the candidate set.

b) The candidate set is then restricted to those ports that satisfy any filters specified. A port is removed from the candidate set if:

   1) The port name appears in the **-exclude_ports** list.

   2) The port is a port on an instance named in the **-exclude_elements** list.

   3) The port direction is not consistent with the direction identified by the **-applies_to** option.

c) The resulting restricted set is the set of ports to be attributed.

If **-model** is specified, the port attributes are applied to the selected ports of each instance of the model. In this case, only names that are declared in the model may be referenced in arguments to this command and all names are interpreted relative to the topmost scope of the model.

-model and **-attribute** can be used together to specify attributes for ports of a hard IP. For example, if ports of the hard IP are connected to each other by the same metal wire, i.e., a feedthrough connection, they should have the **UPF_feedthrough** attribute set to **TRUE**. If a port is not connected to any logic inside the hard IP, it should have the **UPF_unconnected** attribute set to **TRUE**. For more details, see Annex G.

-clamp_value defines the **UPF_clamp_value** attribute, which specifies the clamp value to be used if this port has an isolation strategy applied to it.

-sink_off_clamp defines the **UPF_sink_off_clamp** attribute, which specifies the clamp requirement when the supply set connected to the sink is in a power state with a corresponding simstate of **CORRUPT**.

-source_off_clamp defines the **UPF_source_off_clamp** attribute, which specifies the clamp requirement when the supply set connected to the source is in a power state with a corresponding simstate of **CORRUPT**.

When a user-defined clamp *value* is specified for **UPF_sink_off_clamp** or **UPF_source_off_clamp**, it shall be a legal value for the type of the port. A clamp value of **any** specifies any clamp value legal for the port type is allowed. If the port needs to be isolated in a given context, the specific clamp value to use shall be specified in a **set_isolation** command (see 6.41).

-driver_supply and **–receiver_supply** define the attributes **UPF_driver_supply** or **UPF_receiver_supply**, respectively. These attributes can be used to specify the driver supply of a macro cell output port or the receiver supply of a macro cell input port. They can also be used to specify the driver supply of external logic driving a primary input or to specify the receiver supply of external logic receiving a primary output.

When the **UPF_driver_supply** attribute is defined for a port, it specifies the driver supply of the logic driving the port. If the driving logic is not within the logic design starting at the design root, it is presumed the specified driver supply is the supply for the driver logic; therefore, the port is corrupted when the driver supply is in a simstate other than **NORMAL**. For a port with the attribute **UPF_driver_supply**, when that port has a single source and the driving logic is present within the logic design starting at the design root, it shall be an error if the supply of the driving logic is not the same as, or equivalent to, the specified driver supply.

When the **UPF_receiver_supply** attribute is defined for a port, it specifies the receiver supply of the logic receiving the port. If the receiving logic is not within the logic design starting at the design root, it is presumed the specified receiver supply is the supply for the receiving logic. For a port with the attribute **UPF_receiver_supply**, when that port has a single sink and the receiving logic is present within the logic design, it shall be an error if the supply of the receiving logic is not the same as, or equivalent to, the specified receiver supply.

If **UPF_driver_supply** is not defined for a primary input port or **UPF_receiver_supply** is not defined for a primary output port, the default driver supply or receiver supply, respectively, is an anonymous supply set that is not equivalent to any other supply set.

-pg_type defines the **UPF_pg_type** attribute on a supply port for use with automatic connection semantics. *pg_type_value* is a string denoting the supply port type.

NOTE—**UPF_pg_type** only applies to supply ports and is the only predefined attribute that applies to supply ports. All other attributes apply to logic ports.

If any of **-related_power_port**, **-related_ground_port**, or **-related_bias_ports** is specified, an implicit supply set is created consisting of the supply nets connected to the specified ports. If **-related_power_port** *supply_port_name* and **-related_ground_port** *supply_port_name* are specified, the specified *supply_port_name*s shall be used as the power and ground functions, respectively, of the implicit supply set. If **-related_bias_ports** *supply_port_name_list* is specified, each port in the *supply_port_list* shall have a *pg_type* of `nwell`, `pwell`, `deepnwell`, or `deeppwell`, and each port shall be used as the appropriate bias function of the implicit supply set, as indicated by the value of the associated attribute.

If the port being attributed is `in` mode, the related ports specify the **UPF_receiver_supply** attribute of the port being attributed, as if the implicitly created supply set were specified as the **-receiver_supply** argument. If the port being attributed is `out` mode, the related ports specify the **UPF_driver_supply** attribute of the port being attributed, as if the implicitly created supply set were specified as the **-driver_supply** argument. If the port being attributed is `inout` mode, the related ports specify both the **UPF_receiver_supply** and the **UPF_driver_supply** attributes of the port being attributed, as if the implicitly created supply set were specified as both the **-receiver_supply** and the **-driver_supply** arguments.

By the previous definition, related supplies always refer to the driver and receiver supplies of the logic inside a module.

-feedthrough defines the **UPF_feedthrough** attribute, which identifies a set of ports on the interface of a module or cell that are directly connected to each other inside the module or cell and therefore create a feedthrough through the module or cell.

-unconnected defines the **UPF_unconnected** attribute, which identifies a set of ports on the interface of a module or cell that are not connected to either a source or sink within the module or cell and are not connected to any other port on the interface of the module or cell.

The following also apply:

- It shall be an error if **-model** is specified and **-elements** is also specified.
- It shall be an error if **-related_power_ports**, **-related_ground_ports**, or **-related_bias_ports** is specified, but **-model** is not specified.
- It shall be an error if **-related_ground_port** is specified, but **-related_power_port** is not specified, or if **-related_power_port** is specified, but **-related_ground_port** is not specified.
- It shall be an error if **-related_bias_port** is specified, but either **-related_power_port** or **-related_ground_port** is not specified.
- It shall be an error if a supply port is included in **-ports** and that port has no *pg_type* attribute.
- It shall be an error if **UPF_pg_type** is specified for a port that is not a supply port.
- It shall be an error if no argument is used.

Examples

```
set_port_attributes -ports {my_Logic_Port} -clamp_value 1
OR
set_port_attributes -ports {my_Logic_Port} -attribute {UPF_clamp_value "1"}

set_port_attributes -ports {my_Logic_Port}
    -attribute {UPF_related_power_port "my_VDD"}

set_port_attributes -ports {my_Logic_Port}
    -attribute {UPF_related_ground_port "my_VSS"}

set_port_attributes -ports {my_Logic_Port}
```

```
-attribute {UPF_related_bias_ports "my_VNWELL  my_VPWELL "}
```

The following examples illustrate the use of **set_port_attributes** to specify user-defined attributes of ports, such as attribute values that might be required by tools or verification flows:

```
set_port_attributes -ports {a b c} -attribute {function {power nwell}}
set_port_attributes -ports {a} -attribute {voltage_range {0.0 1.2}}
set_port_attributes -ports {a -attribute {tester_control data}
```

6.47 set_power_switch [deprecated]

This is a deprecated command; see also 6.1 and Annex D.

6.48 set_repeater

Purpose	Specify a repeater (buffer) strategy		
Syntax	**set_repeater** *strategy_name* **-domain** *domain_name* [**-elements** *element_list*] [**-exclude_elements** *exclude_list* [**-source** <*source_domain_name* | *source_supply_ref*>] [**-sink** <*sink_domain_name* | *sink_supply_ref*>] [**-use_equivalence** [<**TRUE** | **FALSE**>]] [**-applies_to** <**inputs** | **outputs** | **both**>] [**-repeater_supply_set** *supply_set_ref*] [**-name_prefix** *string*] [**-name_suffix** *string*] [**-instance** **{**{*instance_name port_name*}*}] [**-update**]		
Arguments	*strategy_name*	The name of the repeater strategy.	
	-domain *domain_name*	The domain for which this strategy is defined.	
	-elements *element_list*	A list of instances or ports to which the strategy potentially applies.	**R**
	-exclude_elements *exclude_list*	A list of instances or ports to which the strategy does not apply.	**R**
	-source <*source_domain_name* | *source_supply_ref*>	The rooted name of a supply set or power domain. When a domain name is used, it represents the primary supply of the specified domain.	**R**
	-sink <*sink_domain_name* | *sink_supply_ref*>	The rooted name of a supply set or power domain.When a domain name is used, it represents the primary supply of the specified domain.	**R**
	-use_equivalence [<**TRUE** | **FALSE**>]	Indicates whether to consider supply set equivalence. If **-use_equivalence** is not specified at all, the default is **-use_equivalence TRUE**; if **-use_equivalence** is specified without a value, the default value is **TRUE**.	**R**
	-applies_to <**inputs** | **outputs** | **both**>	A filter that restricts the strategy to apply only to ports of a given direction.	**R**
	-repeater_supply_set *supply_set_ref*]	The supply set that powers the inserted buffer.	**R**

Arguments	**-name_prefix** *string*] [**-name_suffix** *string*]	The name format (prefix and suffix) for inserted buffer cell instances or nets related to implementation of the strategy.	**R**
	-instance {{*instance_name port_name*}*}	The name of a technology library leaf-cell instance and the name of the logic port that it buffers.	**R**
	-update	Indicates that this command provides additional information for a previous command with the same *strategy_name* and *domain_name* and executed in the same scope.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **set_repeater** command defines a strategy for inserting repeater cells (buffers) for ports on the interface of a power domain (see 6.17). Repeaters are placed within the domain, driven by input ports of the domain, and driving output ports of the domain.

-domain specifies the domain for which this strategy is defined.

-elements explicitly identifies a set of candidate ports to which this strategy potentially applies. The *element_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *element_list*, it is equivalent to specifying all the ports of the instance in the *element_list*. Any *element_list*s specified on the base command or any updates (see **–update**) of the base command are combined. If **–elements** is not specified in the base command or any update, every port on the interface of the domain is included in the *aggregate_element_list* (see 5.10).

-exclude_elements explicitly identifies a set of ports to which this strategy does not apply. The *exclude_list* may contain rooted names of instances or ports in the specified domain. If an instance name is specified in the *exclude_list*, it is equivalent to specifying all the ports of the instance in the *exclude_list*. Any *exclude_list*s specified on the base command or any updates of the base command are combined into the *aggregate_exclude_list* (see 5.10).

The arguments **-source**, **-sink**, and **-applies_to** serve as filters that further restrict the set of ports to which a given **set_repeater** command applies. The command only applies to those ports that satisfy all of the specified filters.

-source is satisfied by any port that is driven by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

-sink is satisfied by any port that is received by logic powered by a supply set that matches (see **-use_equivalence**) the specified supply set, ignoring any isolation or level-shifting cells that have already been inferred or instantiated from an isolation or level-shifting strategy.

NOTE 1—A port that does not have a driver will never satisfy the **-source** filter. A port that does not have a receiver will never satisfy the **-sink** filter.

-use_equivalence specifies whether supply set equivalence is to be considered in determining when two supply sets match. If **-use_equivalence** is specified with the value *False*, the **-source** and **-sink** filters shall match only the named supply set. Otherwise, the **–source** and **–sink** filters shall match the named supply set or any supply set that is equivalent to the named supply set.

-applies_to is satisfied by any port that has the specified mode. For upper boundary ports, this filter is satisfied when the direction of the port matches. For lower boundary ports, this filter is satisfied when the

inverse of the direction of the port matches. For example, a lower boundary port with a direction `OUT` would satisfy the `-applies_to IN` filter, because an output from a lower boundary port is an input to this domain. **-applies_to** is always relative to the specified domain.

The *effective_element_list* (see 5.10) for this command consists of all the port names in the *aggregate_element_list* that are not also in the *aggregate_exclude_list* and that satisfy all of the filters specified in the command. If a port in the *effective_element_list* is not on the interface of the specified domain, it shall not be buffered.

If a given port name is referenced in the *effective_element_list* of more than one repeater strategy of a given domain, the precedence rules (see 5.8) determine which of those strategies actually apply to that port name. If the precedence rules identify multiple strategies that apply to the same port name, then the port name shall be the name of an input port to the domain, and each of those strategies shall each have a **–sink** filter that matches the receiving supply of a different sink domain for the specified input port. It shall be an error if the precedence rules identify multiple strategies that apply to the same port name and that port is an output port of the domain, or more than one strategy applies to the same sink domain for that port.

-repeater_supply_set is implicitly connected to the primary or backup supply ports of the buffer cell. If **-repeater_supply_set** is not specified, then if the primary supply set of the domain containing the driver of the repeater is available in the power domain where the repeater will be located, that supply is used as the default supply. It shall be an error if **repeater_supply_set** is not specified and the default supply is not available in the domain.

-name_prefix specifies the substring to place at the beginning of any generated name implementing this strategy.

-name_suffix specifies the substring to place at the end of any generated name implementing this strategy.

-instance specifies that the repeater functionality exists in the HDL design and *instance_name* denotes the instance providing part or all of this functionality. An *instance_name* is a simple name or a hierarchical name rooted in the current scope. If an empty string appears as an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance should not be reinferred or reimplemented by subsequent tool runs.

–update adds information to the base command executed in the same scope. When specified with **-update**, **-elements** and **-exclude_elements** are additive: the set of instances or ports in the *aggregate_elements_list* is the union of all **–elements** specifications given in the base command and any update of this command, and the *aggregate_exclude_list* is the union of all **–exclude_elements** specifications given in the base command and any update of this command.

The following also apply:

— This command never applies to inout ports.

— The simstate semantics of the repeater supply set apply to the output of a repeater.

NOTE 2—To specify a repeater strategy for a port `P` on the lower boundary of a power domain `D` (see 4.3.1), a **set_repeater** command can specify `-domain D` and specify the port name `I/P`, where `I` is the hierarchical name of an instance that is instantiated in domain `D` but is not in the extent of domain `D`, and `P` is the simple name of the port of that instance. The combination of the **–domain** specification and the hierarchical port name makes it clear this reference is to the HighConn of the specified port, which is part of the lower boundary of the domain `D`.

NOTE 3—Insertion of a repeater may change the driver supply and receiver supply of ports that are sinks or sources, respectively, of the inserted repeater. Such changes may affect the interpretation of **-source** or **-sink** filters of **set_isolation** (see 6.41) or **set_level_shifter** (see 6.43) strategies that apply to those ports. These changes may also affect the default for the input supply set or the output supply set of **set_level_shifter** strategies that apply to those ports.

NOTE 4—The *exclude_list* in **-exclude_elements** can specify instances or ports that have not already been explicitly or implicitly specified via an explicit or implied *element_list*.

Syntax example:

```
set_repeater feedthrough_buffer1
-domain PD3 -applies_to outputs
```

6.49 set_retention

Purpose	Specify a retention strategy		
Syntax	**set_retention** *retention_name* **-domain** *domain_name* [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-retention_supply_set** *ret_supply_set*] [**-no_retention**] [**-save_signal** {*logic_net* <**high** | **low** | **posedge** | **negedge**>} **-restore_signal** {*logic_net* <**high** | **low** | **posedge** | **negedge**>}] [**-save_condition** {*boolean_expression*}] [**-restore_condition** {*boolean_expression*}] [**-retention_condition** {*boolean_expression*}] [**-use_retention_as_primary**] [**-parameters** {<**RET_SUP_COR** | **NO_RET_SUP_COR** | **SAV_RES_COR** | **NO_SAV_RES_COR**> *}] [**-transitive** [<**TRUE** | **FALSE**>]] [**-instance** {{*instance_name* [*signal_name*]}*}] [**-update**] [**-retention_power_net** *net_name*] [**-retention_ground_net** *net_name*]		
Arguments	*retention_name*	Retention strategy name.	
	-domain *domain_name*	The domain for which this strategy is applied.	
	-elements *element_list*	The **-elements** option specifies a list of objects: instances, *retention_list_name* of elements lists (see 6.51), named processes, or sequential `reg` or signal names to which this strategy is applied.	**R**
	-exclude_elements *exclude_list*	The **-exclude_elements** option specifies a list of objects: instances, named processes, or sequential `reg` or signal names that are not included in this strategy.	**R**
	-no_retention	Prevents the inference of retention cells on the specified elements regardless of any other specifications.	**R**
	-retention_supply_set *ret_supply_set*	This option specifies the supply set used to power the logic inferred by the *retention_name* strategy.	**R**
	-save_signal {*logic_net* <**high** | **low** | **posedge** | **negedge**>} **-restore_signal** {*logic_net* <**high** | **low** | **posedge** | **negedge**>}	The **-save_signal** and **-restore_signal** options specify a rooted name of a logic net or port and its active level or edge.	**R**
	-save_condition {*boolean_expression*}	The **-save_condition** option specifies a Boolean expression (see 5.4). The default is *True* if the **-save_signal**/**-restore_signal**s are specified, else the **-save_condition** is a don't care.	**R**

Arguments	**-restore_condition** {*boolean_expression*}	The **-restore_condition** option specifies a Boolean expression. The default is *True* if the **-save_signal**/**-restore_signal**s are specified, else the **-restore_condition** is a don't care.	**R**
	-retention_condition {*boolean_expression*}	The **-retention_condition** option specifies a Boolean expression. The default is *True* if the **-save_signal**/**-restore_signal**s are specified, else the default value of **-retention_condition** is *False*.	**R**
	-use_retention_as_primary	The **-use_retention_as_primary** option specifies that the storage element and its output are powered by the retention supply.	**R**
	-parameters {<**RET_SUP_COR** | **NO_RET_SUP_COR** | **SAV_RES_COR** | **NO_SAV_RES_COR**> *}	The **-parameters** option provides control over retention register corruption semantics.	**R**
	-transitive [<**TRUE** | **FALSE**>]	If **-transitive** is not specified at all, the default is **-transitive TRUE**. If **-transitive** is specified without a value, the default value is **TRUE**.	
	-instance {{*instance_name* [*signal_name*]}*}	The name of a technology library leaf-cell instance and the optional name of the signal that it retains. If this instance has any unconnected supply ports or save and restore control ports, then these ports need to have identifying attributes in the cell model, and the ports shall be connected in accordance with this **set_retention** command.	**R**
	-update	Use **-update** if the *retention_name* has already been defined.	**R**
Legacy arguments	**-retention_power_net** *net_name*	This option defines the supply net used as the power for the retention logic inferred by this strategy. This is a legacy option; see also 6.1 and Annex D.	**R**
	-retention_ground_net *net_name*	This option defines the supply net used as the ground for the retention logic inferred by this strategy. This is a legacy option; see also 6.1 and Annex D.	**R**
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.		

The **set_retention** command specifies a set of objects in the domain that need to be retention registers and identifies the save and restore behavior. If an instance is specified, all registers within the instance acquire the specified retention strategy. If a process is specified, all registers inferred by the process acquire the specified retention strategy. If a `reg`, signal, or variable is specified and that object is a sequential element, the implied register acquires the specified retention strategy. Any specified `reg`, signal, or variable that does not infer a sequential element shall not be changed by this command.

If **-elements** is specified, only elements in the element list that are also a part of the *domain_name* are included. Any element names outside the extent of *domain_name* are excluded. When **-elements** is not specified, this is equivalent to using the elements list that defines the power domain. When used with **-update**, **-elements** is additive such that the set of elements or signals is the union of all calls of this command for a given strategy specifying any of these parameters.

-exclude_elements can also be used to define a list of storage elements that are not included in this strategy. When used with **-update**, **-exclude_elements** is additive such that the set of elements or signals excluded is the union of all calls of this command for a given strategy.

-retention_supply_set powers the register holding the retained value. After the strategy has been completely applied, it shall be an error if the retention supply set is not defined for a strategy and the domain does not have a default *ret_supply_set*.

For a balloon-style retention register (see 4.3.4), the retained value is transferred to the register on the restore event when **-restore_condition** evaluates to *True*. The restore event is the rising or falling edge of an edge-triggered restore event or the trailing edge of a level-sensitive restore event. A level-sensitive restore event has priority over any other register operation.

-restore_condition gates the restore event, defining the restore behavior of the register. If the **-save_signal**/**restore_signal**s are not specified, the **-restore_condition** becomes a don't care. The register is restored when the restore event occurs and the **-restore_condition** is *True*.

For a balloon-style retention register, the retained value shall be the register's value at the time of the save event when **-save_condition** evaluates to *True*. The save event is the rising or falling edge of an edge-triggered save event or the trailing edge of a level-sensitive save event.

-save_condition gates the save event, defining the save behavior of the register. If the **-save_signal**/**restore_signal**s are not specified, the **-save_condition** becomes a don't care. The register contents are saved when the save event occurs and the **-save_condition** is *True*.

-retention_condition defines the retention behavior of the retention element while the primary supply is not `NORMAL`. If the retention condition evaluates to `FALSE` and the primary supply is not `NORMAL`, the receiving supply of any pin listed in the **-retention_condition** shall be assumed to be the retention supply of the retention strategy.

-save_condition, **-restore_condition**, and **-retention_condition** shall only reference logic nets or ports rooted in the current scope. The **-save_signal**/**-restore_signal**/**-save_condition**/**-restore_condition** apply only to balloon-style retention registers. For master-/slave-alive implementations (see 4.3.4), the **-save_signal**/**-restore_signal** should not be specified. The retention behavior of this style is specified through the **-retention_condition**. It shall be an error if **-save_signal**/**-restore_signal** is not specified and the **-retention_condition** is also not specified.

-use_retention_as_primary powers the storage element and the output drivers of the register using the retention supply. The result of this is the simstate for the retention supply set is applied to the register's output. Inferred state elements shall be consistent with the **-use_retention_as_primary** constraint.

NOTE 1—UPF only supports the output pins' driving supply being different from the primary supply (with **-use_retention_supply_as_primary**); the input pins' receiving supply can only be assumed to be the primary supply of the domain.

NOTE 2—The **-use_retention_as_primary** changes the driver supply of ports that are sinks of the inserted retention register. Such changes may affect the interpretation of the **-source** filters of the **set_repeater** (see 6.48), **set_isolation** (see 6.41), or **set_level_shifter** (see 6.43) strategies that apply to those ports.

The **-parameters** option provides control over retention register corruption semantics. For a retention strategy, it is an error to specify:

— both **RET_SUP_COR** and **NO_RET_SUP_COR**; or

— both **SAV_RES_COR** and **NO_SAV_RES_COR**.

RET_SUP_COR activates and **NO_RET_SUP_COR** deactivates corruption of the normal mode register when retention supplies are **CORRUPT**. When neither value is specified for a retention strategy, **RET_SUP_COR** is the default value.

SAV_RES_COR activates and **NO_SAV_RES_COR** deactivates corruption of the normal mode register during concurrent assertion of level-sensitive **save**, **save_condition**, **restore**, and **restore_condition**. When neither value is specified for a retention strategy, **SAV_RES_COR** is the default value.

-instance specifies that the retention functionality exists in the HDL design and *instance_name* denotes the instance providing part or all of this functionality. An *instance_name* is a hierarchical name rooted in the current scope. If an empty string appears in an *instance_name*, this indicates that an instance was created and then optimized away. Such an instance should not be reinferred or reimplemented by subsequent tool runs.

–update adds information to the base command executed in the same scope of the power domain for which the inferred cells are defined.

The elements requiring retention can be attributed in HDL as shown in 6.51.

For details on the simulation semantics of this command, please refer to 9.6.

Examples

Some examples of the **set_retention** command are shown as follows:

a) Save-restore balloon-type RFF

   Has an explicit save and restore pin, which perform save/restore functions.

```
set_retention my_ret \
-save_signal {save high} \
-restore_signal {restore high} \
 ...
```

b) Single retention pin balloon-type RFF

   1) Has single pin that performs save/restore functions.
   2) To remain in a retention state, the retention pin shall be kept at a certain value.

```
set_retention my_ret \
-save_signal {ret posedge} \
-restore_signal {ret negedge} \
-retention_condition {ret} \
...
```

c) Single retention pin slave-alive type RFF

   1) Has a single retention control pin, but no save/restore is involved as the slave latch (or storage element) is powered by the retention supply.
   2) Requires the retention pin to remain at a certain value to be in retention mode.

```
set_retention my_ret \
-retention_condition {ret} \
...
```

   NOTE—No save/restore signals/conditions are specified in this case. Here, the retention condition is explicitly specified, meaning the retention condition has to be true during retention mode.

d) No pin slave alive type RFF with output powered by retention supply

   1) Has no control pin and no save/restore is involved as the slave latch (or storage element) is powered by the retention supply.
   2) Requires the clocks/async resets to be related to retention supply and parked at a certain value during retention mode.
   3) The **-use_retention_as_primary** is specified as the output is expected to be powered by the retention supply.

```
set_retention my_ret \
-retention_condition {!clock && nreset} \
-use_retention_as_primary \
...
```

6.50 set_retention_control [deprecated]

This is a deprecated command; see also 6.1 and Annex D. To model mutex assertions using **bind_checker**, see 6.9.

6.51 set_retention_elements

Purpose	Create a named list of elements to be used in **set_retention** or **map_retention_cell** commands	
Syntax	**set_retention_elements** *retention_list_name* **-elements** *element_list* [**-applies_to** <**required** | **not_optional** | **not_required** | **optional**>] [**-exclude_elements** *exclude_list*] [**-retention_purpose** <**required** | **optional**>] [**-transitive** [<**TRUE** | **FALSE**>]] [**-expand** [<**TRUE** | **FALSE**>]]	
Arguments	*retention_list_name*	A simple name; this shall be unique within the current *scope*.
	-elements *element_list*	A list of rooted names: instances, named processes, sequential `regs`, or signal names.
	-applies_to <**required** | **not_optional** | **not_required** | **optional**>	Filter elements based on the **UPF_retention** attribute value.
	-exclude_elements *exclude_list*	A list of rooted names: instances, named processes, sequential `regs`, or signal names.
	-retention_purpose <**required** | **optional**>	The intended retention use of *retention_list_name*. The default is **required**.
	-transitive [<**TRUE** | **FALSE**>]	If **-transitive** is not specified at all, the default is **-transitive TRUE**. If **-transitive** is specified without a value, the default value is **TRUE**.
Deprecated arguments	**-expand** [<**TRUE** | **FALSE**>]	When **-expand** is **TRUE** (the default), elements are expanded as though every register that otherwise would be included had been specified directly in *element_list*. This is a deprecated option; see also 6.1 and Annex D.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **set_retention_elements** command defines a “atomic” list of objects whose state shall be retained or not retained together by the **set_retention** and **map_retention_cell** commands (see 6.49 and 6.33).

If the state of any element in *retention_element_list* is retained, the state of every element in *retention_element_list* shall be retained.

-applies_to filters the *effective_element_list,* removing any elements that do not have a **UPF_retention** attribute value consistent with the selected filter choice: **required**, **not_optional**, **not_required**, or **optional**, as follows:

required matches all elements that have the **UPF_retention** attribute value `required`.

optional matches all elements that have the **UPF_retention** attribute value `optional`.

not_required matches all elements that do not have the **UPF_retention** attribute value `required`.

not_optional matches all elements that do not have the **UPF_retention** attribute value `optional`.

When **-retention_purpose** is **required**, retention shall only be necessary if elements in the *retention_element_list* are in the extent of a power domain that has retained elements.

It shall be an error if an element belonging to *retention_element_list* is not retained when any element in the same *retention_element_list* is retained.

It shall be an error if **retention_purpose** is **required** and an element belonging to *retention_element_list* is not retained when any element in the same power domain extent is retained.

Syntax example:

```
set_retention_elements ret_chk_list
   -elements {proc_1 sig_a}
```

6.52 set_scope

Purpose	Specify the current scope	
Syntax	**set_scope** *instance*	
Arguments	*instance*	The instance that becomes the current scope upon completion of the command.
Return value	Return the current scope prior to execution of the command as a design-relative hierarchical name (see 5.3.3.3) if successful or raise a `TCL_ERROR` if it fails (e.g., if the instance does not exist).	

The **set_scope** command changes the current scope to the specified scope and returns the name of the previous scope as a design-relative hierarchical name.

The following also apply:

- The instance name may be a simple name, a scope-relative hierarchical name, a design-relative hierarchical name, the symbol `/`, the symbol `.`, or the symbol `...`
- If the instance name is `/`, the current scope is set equal to the current design top instance.
- If the instance name is `.`, the current scope is unchanged.
- If the instance name is `..`, and the current scope is not equal to the current design top instance, the current scope is changed to the parent scope.
- It is an error if the instance name is `..` and the current scope is equal to the current design top instance.

Examples

Given the hierarchy

```
top/
   mid/
      bot/
```

if the current design top instance is `/top`, and the current scope is `/top/mid`, then

```
set_scope bot ;# changes current scope to /top/mid/bot (child of current scope)
set_scope . ;# leaves current scope unchanged as /top/mid (current scope)

set_scope .. ;# changes current scope to /top (parent of current scope)
set_scope / ;# changes current scope to /top (current design top instance)
```

If the current design top instance is `/top/mid` and the current scope is `/top/mid`, then

```
set_scope bot ;# changes current scope to /top/mid/bot
set_scope . ;# leaves current scope unchanged as /top/mid
set_scope .. ;# results in an error
set_scope / ;# changes current scope to /top/mid (current design top instance)
```

If the current design top instance is `/top` and the current scope is `/top`, then

```
set_scope mid/bot ;# changes current scope to /top/mid/bot
set_scope . ;# leaves current scope unchanged as /top
set_scope .. ;# results in an error
set_scope / ;# changes current scope to /top (current design top instance)
```

6.53 set_simstate_behavior

Purpose	Specify the simulation simstate behavior for a model or library	
Syntax	**set_simstate_behavior <ENABLE | DISABLE>** [**-lib** *name*] [**-model** *model_list*] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*]	
Arguments	<**ENABLE | DISABLE**>	Define if the UPF simstate behavior shall be enabled for the specified model(s).
	-lib *name*	The library name.
	-model *model_list*	One or more model names.
	-elements *element_list*	A list of instances.
	-exclude_elements *exclude_list*	A list of instances to exclude from the *effective_element_list* (see 5.10).
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

This command specifies the simstate behavior for models or instances.

If **ENABLE** is specified, the simstate simulation semantics are applied for every supply set automatically connected to an instance of the model. See also 9.4.

a) If there is a single supply set connected, the simstates for that supply set are applied.

b) When no supply set is connected, and each port to which a supply net is connected is of a different *pg_type*, an anonymous supply set is created containing the supply nets connected to each port, with each supply net associated with the function appropriate for the *pg_type* of that port, and the default simstates for that supply set are applied for the model.

c) When there are multiple supply sets connected, the simstates of all supply sets are applied.

d) For a hard macro instance in which there are multiple supply pins of the same *pg_type*, an anonymous supply set is created for each unique combination of supply pins identified as related supplies of a logic pin of the macro instance, with each supply pin associated with the function appropriate for the *pg_type* of that pin. The default simstates of each supply set are applied during simulation for any logic pin related to that supply set.

e) For an instance of a hard macro behavioral model, each logic pin of the instance is corrupted according to the applicable simstate of the supply set associated with the logic pin.

If **-model** is not defined and **-lib** is specified, the simstate behavior is defined for all models in *name*.

It shall be an error if

— **-model** is specified and any of the model(s) cannot be found.
— **-elements** is specified and any of the element(s) cannot be found.
— **-exclude_elements** is specified and any of the *exclude_elements*(s) cannot be found.
— **-exclude_elements** is specified and **-model**, **-elements**, or **-lib** is not specified.
— A given model has its simstate behavior both enabled and disabled, by **set_simstate_behavior** commands, **UPF_simstate_behavior** attributes, or a combination thereof.
— *effective_element_list* is empty.

Simstate behavior of a module can be enabled or disabled in HDL using the following attributes:

Attribute name: **UPF_simstate_behavior**

Attribute value: <**"ENABLE"** | **"DISABLE"**>

SystemVerilog or Verilog-2005 example:

```
(* UPF_simstate_behavior = "ENABLE" *) module my_adder;
```

VHDL example:

```
attribute UPF_simstate_behavior of my_adder : entity is
"ENABLE";
```

Syntax example:

```
set_simstate_behavior ENABLE -lib library1 -model ANDX7_non_power_aware
```

6.54 upf_version

Purpose	Retrieves the version of UPF being used to interpret UPF commands and documents the UPF version for which subsequent commands are written
Syntax	**upf_version** [*string*]
Arguments	*string* The UPF version for which subsequent commands are written.

Return value	Returns the version of UPF currently being used to interpret UPF commands.

The **upf_version** command returns a string value representing the UPF version currently being used by the tool reading the UPF file. When the UPF version defined by this standard is being used, the returned value shall be the string "2.1". **upf_version** may also include an argument that documents the UPF version for which the UPF commands that follow were written. For UPF commands intended to be interpreted according to the UPF version defined by this standard, the argument shall be the string "2.1".

This standard does not define any other value for the returned value of the **upf_version** command or for the *string* argument. This standard also does not define how a tool uses the specified UPF version argument; in particular, this standard does not define the meaning of a description consisting of UPF commands intended to be interpreted according to different UPF versions.

Syntax example:

```
upf_version 2.1
```

6.55 use_interface_cell

Purpose	Specify the functional model and a list of implementation targets for isolation and level-shifting	
Syntax	**use_interface_cell** *interface_implementation_name* **-strategy** *list_of_isolation_level_shifter_strategies* **-domain** *domain_name* **-lib_cells** *lib_cell_list* [**-port_map** {{*port net_ref*}*}] [**-elements** *element_list*] [**-exclude_elements** *exclude_list*] [**-applies_to_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>] [**-update_any** <**0** | **1** | **known** | **Z** | **latch** | *value*>] [**-force_function**] [**-inverter_supply_set** *list*]	
Arguments	*interface_implementation_name*	The interface cell implementation strategy.
	-strategy *list_of_isolation_level_shifter_strategies*	The isolation or level-shifter strategy, or a pair of isolation and level-shifter strategies, as defined by **set_isolation** and **set_level_shifter**.
	-domain *domain_name*	The domain in which the strategies are defined.
	-lib_cells *lib_cell_list*	A list of library cell names.
	-port_map {{*port net_ref*}*}	The *port* and the net (*net_ref)* connections.
	-elements *element_list*	A list of ports from the *list_of_isolation_level_shifter_strategies* to which the command applies.
	-exclude_elements *exclude_list*	A list of ports from the *list_of_isolation_level_shifter_strategies* to which this command does not apply.

Arguments	**-applies_to_clamp** <**0** | **1** | **any** | **Z** | **latch** | *value*>	Only ports that have the specified clamp value are mapped.
	-update_any <**0** | **1** | **known** | **Z** | **latch** | *value*>	What is now the clamp value when **-applies_to_clamp** is **any**.
	-force_function	The first model in *lib_cell_list* is used as the functional specification of isolation behavior.
	-inverter_supply_set *list*	The supply set implicitly connected to any inversion logic required by an isolation signal connection.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **use_interface_cell** command provides user control for the integration of isolation and level-shifting. The command specifies the implementation choices through **-lib_cells** and the functional isolation behavior to be used if **-force_function** is specified.

Each cell specified in **-lib_cells** shall be defined by a **define_isolation_cell** (see 7.4) or **define_level_shifter_cell** (see 7.5) command or defined in the Liberty file with required attributes.

NOTE—Unlike **map_isolation_cell** and **map_level_shifter_cell, use_interface_cell** can be used to manually map any of isolation, level-shifting, or combined isolation level-shifting cells. It may apply to an isolation strategy, a level-shifting strategy, or one of each.

When **-force_function** is specified the first model in *lib_cell_list* shall be used as the functional model. The isolation sense specification for the isolation strategy is ignored when **-force_function** is specified. It is erroneous if the functional model clamps to a value that is different to the previously specified port clamp value.

-elements selects the ports from the specified list of strategies to which the mapping command is applied. If **-elements** is not specified, all ports inferred from the list of strategies shall have the mapping applied. When **-applies_to_clamp** is specified, this command is applied only to the ports with that clamp value.

When **-applies_to_clamp** is **any**, **-update_any** shall be used to specify the clamp value after mapping. An **-update_any** value of **known** specifies that the isolation function is more complex than can be specified by a single value.

-port_map connects the specified *net_ref* to a *port* of the model. A *net_ref* may be one of the following:

a) A logic net name

b) A supply net name

c) One of the following symbolic references

   1) **isolation_supply_set.***function_name*

      function_name refers to the supply net corresponding to the function it provides to the **isolation_supply_set**.

   2) **isolation_supply_set**[*index*]**.***function_name*

      i) *index* is a non-negative integer corresponding to the position in the **isolation_supply_set** list specified for the isolation strategy.

      ii) The **isolation_supply_set** *index* shall be specified if the isolation strategy specified more than one **isolation_supply_set**.

3) **isolation_signal**

   i) Refers to the isolation signal specified in the corresponding isolation strategy.

   ii) To invert the sense of the isolation signal the Verilog bit-wise negation operator ~ can be specified before the isolation_signal. The logic inferred by the negation shall be implicitly connected to the **inverter_supply_set** if specified, otherwise the **isolation_supply_set** shall be used.

4) **isolation_signal**[*index*]

   i) *index* is a non-negative integer corresponding to the position in the **isolation_signal** list specified for the isolation strategy.

   ii) The **isolation_signal** *index* shall be specified if the isolation strategy specified more than one **isolation_signal**.

   iii) To invert the sense of the isolation signal the Verilog bit-wise negation operator ~ can be specified before the isolation_signal. If the **isolation_signal** is being inverted then the **inverter_supply_set[***index***]** if specified shall be implicitly connected to the inferred inverter, otherwise the **isolation_supply_set**[*index*] shall be used.

5) **input_supply_set.***function_name*

   function_name refers to the supply net corresponding to the function it provides to the level-shifter **input_supply_set**.

6) **output_supply_set.***function_name*

   function_name refers to the supply net corresponding to the function it provides to the level-shifter **output_supply_set**.

7) **internal_supply_set.***function_name*

   function_name refers to the supply net corresponding to the function it provides to the level-shifter **internal_supply_set**.

The **-port_map** option shall not reference the data input port or the data output port. The input port shall be connected to the data input for the interface cell and the output port connected to the data output for the interface cell.

It shall be an error if

- *domain_name* does not indicate a previously created power domain.
- A port in the *port_list* is not covered by a **set_isolation** command.
- *list_of_isolation_level_shifter_strategies* is an empty list.
- **-force_function** is not specified and none of the specified models in *lib_cell_list* implements the functionality specified by the corresponding *isolation_strategy* and port attributes.
- **-update_any** is specified and **-applies_to_clamp** is not **any**.
- After completing the *port* and *net_ref* connections and the data input and output connections, any port is unconnected.
- Ports specified by **-elements** are not included in all specified strategies.
- More than one isolation strategy is specified.
- More than one level-shifter strategy is specified.

Syntax example:

```
use_interface_cell my_interface -strategy {ISO1 LS1} -domain PD1\
-elements {top/moduleA/port1 top/moduleA/port2 top/moduleA/port3}
```

7. Power management cell commands

7.1 Introduction

This clause documents the syntax for each UPF power management cell command. A power management cell is one of the following:

- "Always-on" cell
- Diode clamp
- Isolation cell
- Level-shifter cell
- Power-switch cell
- Retention cell

Power management cell commands define characteristics of the instances of power management cells used to implement and verify the power intent for a given design. These commands do not alter the existing library cell definitions and only have semantics when they are used with design power intent commands (see Clause 6).

Similar to how libraries are processed in a design flow, UPF power management cell commands need to be processed before any other power intent commands and after the relevant cell libraries have been loaded.

It is an error if conflicting information is specified in multiple commands (of any type).

To understand the relationship between each UPF power management cell command and its library cell definition in Liberty format, see Annex H.

7.2 define_always_on_cell

Purpose	Identify always-on cells	
Syntax	**define_always_on_cell** **-cells** *cell_list* **-power** *pin* **-ground** *pin* [**-power_switchable** *pin*] [**-ground_switchable** *pin*] [**-isolated_pins** *list_of_pin_lists* [**-enable** *expression_list*]]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as always-on cells.
	-power *pin*	Identifies the power pin of the cell. If this option is specified with the **-power_switchable** option, it indicates this is a non-switchable power pin.
	-ground *pin*	Identifies the ground pin of the cell. If this option is specified with the **-ground_switchable** option, it indicates this is a non-switchable ground pin.
	-power_switchable *pin*	Specifies the power pin to be connected via a rail connection to the switchable power supply.
	-ground_switchable *pin*	Specifies the ground pin to be connected via a rail connection to the switchable ground supply.
	-isolated_pins *list_of_pin_lists*	Specifies a list of pin lists. Each pin list groups pins that are isolated internally with the same isolation control signal. These pin lists can only contain input pins.
	-enable *expression_list*	Specifies a list of simple expressions. Each simple expression describes the isolation control condition for the corresponding isolated pin list in the **-isolated_pins** option. If the internal isolation does not require a control signal, use an empty string for that pin list. The number of elements in this list shall correspond to the number of lists specified for the **-isolated_pins** option.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_always_on_cell** library command identifies the library cells having more than one set of power and ground pins that can remain functional even when the supply to the switchable power or ground pin is switched off.

NOTE—Although a cell is called *always-on* does not mean the cell can never be powered off. When the supply to non-switchable power or ground of such cell is switched off, the cell becomes non-functional. In other words, the term always-on actually means r*elative always-on*.

By default, all input and output pins of this cell are related to the non-switchable power and ground pins.

Examples

The following example defines cell `aon_cell` as an always-on cell. The cell had three isolated pins: `pin1`, `pin2`, and `pin3`. Pins `pin1` and `pin2` have the same isolation control signal iso1, but `pin3` has no isolation control signal.

```
define_always_on_cell -cells aon_cell
   -isolated_pins { {pin1 pin2} {pin3}} -enable {!iso1 ""}
```

The following example defines cell `AND2_AON` as an always-on cell. The cell has two power pins and performs the `AND` function (as defined in the library) as long as the supply connected to power pin `VDD` is not switched off.

```
define_always_on_cell -cells AND2_AON -power_switchable VDDSW
   -power VDD -ground VSS
```

7.3 define_diode_clamp

Purpose	Identify diode cells or cells pins with diode protection	
Syntax	**define_diode_clamp** **-cells** *cell_list* **-data_pins** *pin_list* [**-type** <**power** | **ground** | **both**>] [**-power** *pin*] [**-ground** *pin*]	
Arguments	**-cells** *cell_list*	Identifies cells as diode clamp cells or pins of the specified cells as diode clamp pins.
	-data_pins *pin_list*	Specifies a list of cell input pins that have built-in clamp diodes.
	-type <**power** | **ground** | **both**>	Specifies the type of clamp diode associated with the data pins. The type determines whether to use the power pin, ground pin, or both. Possible values are as follows: **both** indicates a power and ground clamp diode **ground** indicates a ground clamp diode **power** indicates a power clamp diode (the default)
	-power *pin*	Specifies the cell pin that connects to the power net.
	-ground *pin*	Specifies the cell pin that connects to the ground net.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_diode_clamp** library command identifies a list of library cells that are power cells, ground cells, or power and ground diode clamp cells, or complex cells that have input pins with built-in clamp diodes.

When **-type** is **ground**, then **-power** is optional. When **-type** is power, then **-ground** is optional. When **-type** is **both**, then both **-power** and **-ground** need to be specified as well.

It shall be an error if neither **-power** nor **-ground** is specified.

NOTE—The **define_diode_clamp** command is typically used for pins that have antenna protection diodes. Hence, this command may apply to regular non-power managed cells.

Examples

The following command defines a cell `cellA` with diode protection at the pin `in1` where the diode is connected to the power pin `VDD1` of the cell.

```
define_diode_clamp -cells cellA -data_pins in1 -type power -power VDD1
```

7.4 define_isolation_cell

Purpose	Identify isolation cells	
Syntax	**define_isolation_cell** **-cells** *cell_list* [**-power** *power_pin*] [**-ground** *power_pin*] {**-enable** *pin* [**-clamp_cell** <**high** | **low**>] | **-pin_groups** **{{***input_pin output_pin* [*enable_pin*]**}*}** | **-no_enable** <**high** | **low** | **hold**>} [**-always_on_pins** *pin_list*] [**-aux_enables** *ordered_pin_list*] [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*] [**-valid_location** <**source** | **sink** | **on** | **off** | **any**>] [**-non_dedicated**]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as isolation cells.
	-power *power_pin*	Identifies the power pin of the cell. If this option is specified with the **-power_switchable** option for a multi-rail isolation cell, it indicates this is a non-switchable power pin.
	-ground *power_pin*	Identifies the ground pin of the cell. If this option is specified with the **-ground_switchable** option for a multi-rail isolation cell, it indicates this is a non-switchable ground pin.
	-enable *pin*	Identifies the specified cell pin as the isolation enable pin. For non-clamp type isolation cells, the enable pin polarity is determined by the cell function defined in the library files. For the special clamp type cell identified by the **–clamp_cell** option, the enable polarity is active high if the clamp output is low and the enable polarity is active low if the clamp output is high. For a multi-rail isolation cell, the enable pin is related to the non-switchable power and ground pins of the cells.
	-clamp_cell <**high** | **low**>	Indicates the specified cells are isolation clamp cells. Such a cell, which consists of a single PMOS or NMOS transistor, does not perform any logic function and is only used to clamp a net to **high** or **low** when the enable pin is activated.
	-pin_groups **{{***input_pin output_pin* [*enable_pin*]**}*}**	Specifies a list of input-output paths for multi-bit isolation cells. Each group in the list specifies one cell input pin, one cell output pin, and one optional enable pin that applies to the specified path. An enable pin may appear in more than one group. It is an error if the same input or output pin appears in more than one group.
	-no_enable <**high** | **low** | **hold**>	Specifies the following: The isolation cell does not have an enable pin. The output of the cell when the supply for the switchable power (or ground) pin is powered down. Possible values are as follows: **high** indicates the cell output is logic value `1` **low** indicates the cell output is logic value `0` **hold** indicates the cell output is the same as the logic value before the supply for the switchable power or ground is powered down
	-always_on_pins *pin_list*	Specifies a list of cell pins related to the nonswitchable power and non-switchable ground pins of the cell.
	-aux_enables *ordered_pin_list*	Specifies additional or auxiliary enable pins for the isolation cell.

Arguments	**-power_switchable** *power_pin*	Identifies the switchable power pin of a multi-rail isolation cell.
	-ground_switchable *ground_pin*	Identifies the switchable ground pin of a multi-rail isolation cell.
	-valid_location <**source** | **sink** | **on** | **off** | **any**>	Specifies the valid location of the isolation cell. The default value is **sink**.
	-non_dedicated	Allows the specified cells to be used as normal cells, not for power management purposes.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_isolation_cell** library command identifies the library cells that can be used for isolation in a design with power gating.

By default, the output pin of a multi-rail isolation cell is related to the non-switchable power and ground pins. The non-enable input pin is related to the switchable power and ground pins. A *multi-rail isolation cell* is a cell with two power or ground pins.

If **-clamp_cell** is specified with value **high**, the only supply pin that can be specified is **-power**. If **-clamp_cell** is specified with **low**, the only supply pin that can be specified is **-ground**. For all other isolation cells, both **-power** and **-ground** shall be specified.

The **-aux_enables** option specifies additional or auxiliary enable pins for the isolation cell. By default, all pins specified in this option are related to the switchable power or ground pin. The list is an ordered list and each element can be accessed by using index starting at `1`, where the isolation enable pin specified in the **-enable** option is assumed to be index `0`.

If an auxiliary enable pin is related to the non-switchable power or ground, that pin shall also be specified using the **-always_on_pins** option. The logic that drives this pin shall be on when the isolation enable is asserted at pin specified by the **-enable** option.

The **-valid_location** option specifies the valid location of the isolation cell, as follows:

a) **source**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the driving supply set for a net requiring isolation. Such cells are typically multi-rail isolation cells and used for off-to-on isolation. It typically relies on its switchable power and ground supply for its normal function and on its non-switchable power or ground supply to provide the isolation function. See item d) for **off** value for special cases.

b) **sink**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the receiving supply set for a net requiring isolation. Such cells are typically single-rail isolation cells and used for off-to-on isolation.

c) **on**—indicates the cell can only be inserted in the location where the primary supply set is equivalent to either the driving supply set or the receiving supply for a net requiring isolation and the primary supply set is not switched off when the isolation function is needed. When used for off-to-on isolation, it is equivalent to **sink**. Such cells are typically single-rail isolation cells.

d) **off—**indicates the cell can be inserted in a location where the primary supply set is equivalent to either the driving supply set or the receiving supply for a net requiring isolation and the primary supply set may be switched off when the isolation function is needed. When used for off-to-on isolation, it is equivalent to **source**. Such cells are typically multi-rail isolation cells.

NOTE—Some single-rail isolation cells with special circuit structure can also be used in the switched-off domain. For example, a single-rail `NOR` gate can be placed in a power-switched-off domain for off-to-on isolation with an output value `low`; a single-rail `NAND` gate can be placed in the ground switched-off domain for off-to-on isolation with an output value `high`.

e) **any**—indicates the cell can be placed in any location. Such cells are typically multi-rail isolation cells. In addition, this cell is designed in a way that neither its normal function nor its isolation function relies on the primary supply of the domain it locates.Therefore, this type of cell can be used for off-to-on or on-to-off isolation.

Examples

The following isolation cell can be placed in any location for a design that uses ground switches for shutoff. `VDD` is the rail pin for power connection and `GVSS` is the ground pin for non-switchable ground connection. This cell does not have a rail pin for ground connection.

```
define_isolation_cell -cells iso_cell1 -power VDD -ground GVSS
    -enable iso_en -valid_location any
```

The following examples illustrate the use of the **-pin_groups** option to specify multi-bit isolation cells with two paths:

```
define_isolation_cell -cells mbit_iso1 -pin_groups { { datain1 dataout1
   iso1 } { datain2 dataout2 iso2 } }
   -power VDD -ground VSS -valid_location sink
define_isolation_cell -cells mbit_iso2 -pin_groups { { datain1 dataout1 }
   { datain2 dataout2} }
   -power VDD -ground VSS -valid_location sink
```

For cell `mbit_iso1`, there are two isolation paths. The first is from data input `datain1` to output `dataout1` with `iso1` as the isolation enabler. The second is from data input `datain2` to output `dataout2` with `iso2` as the isolation enabler.

For cell `mbit_iso2`, there are also two isolation paths. However, this special isolation cell has no isolation enabler to control each path. As a result, there is no isolation enable signal defined in each group.

7.5 define_level_shifter_cell

<table>
<tr><td>Purpose</td><td colspan="2">Identify level-shifter cells</td></tr>
<tr><td>Syntax</td><td colspan="2">define_level_shifter_cell
-cells cell_list
[-input_voltage_range {{lower_bound upper_bound}*}]
[-output_voltage_range {{lower_bound upper_bound}*}]
[-ground_input_voltage_range {{lower_bound upper_bound}*}]
[-ground_output_voltage_range {{lower_bound upper_bound}*}]
[-direction <low_to_high | high_to_low | both>]
[-input_power_pin power_pin]
[-output_power_pin power_pin]
[-input_ground_pin ground_pin]
[-output_ground_pin ground_pin]
[-ground ground_pin] [-power power_pin]
[-enable pin | -pin_groups {{input_pin output_pin [enable_pin]}*}]
[-valid_location <source | sink | either | any>]
[-bypass_enable expression] [-multi_stage integer]</td></tr>
<tr><td rowspan="7">Arguments</td><td>-cells cell_list</td><td>Identifies the specified cells as level-shifter cells.</td></tr>
<tr><td>input_voltage_range {{lower_bound upper_bound}*}</td><td>Identifies a list of voltage ranges for the input (source) supply voltage that can be handled by this level-shifter.
The voltage range shall be specified as follows:
<code>{lower_bound upper_bound}</code>
This option should only be specified for power-shifting cells.</td></tr>
<tr><td>-output_voltage_range {{lower_bound upper_bound}*}</td><td>Identifies a list of voltage ranges for the output (destination) power supply voltage that can be handled by this level-shifter.
The voltage range shall be specified as follows:
<code>{lower_bound upper_bound}</code>
This option should only be specified for power-shifting cells.</td></tr>
<tr><td>-ground_input_voltage_range {{lower_bound upper_bound}*}</td><td>Identifies a list of voltage ranges for the input (source) ground supply voltage that can be handled by this level-shifter.
The voltage range shall be specified as follows:
<code>{lower_bound upper_bound}</code>
This option should only be specified for ground-shifting cells.</td></tr>
<tr><td>-ground_output_voltage_range {{lower_bound upper_bound}*}</td><td>Identifies a list of voltage ranges for the output (destination) ground supply voltage that can be handled by this level-shifter.
The voltage range shall be specified as follows:
<code>{lower_bound upper_bound}</code>
This option should only be specified for ground-shifting cells.</td></tr>
<tr><td>-direction <low_to_high | high_to_low | both></td><td>Specifies whether the level-shifter can be used between a driver with lower voltage swing and a receiver with higher voltage swing (low_to_high), or vice versa (high_to_low), or both (both). The <i>voltage swing</i> is simply the difference between the power voltage and ground voltage. The default is low_to_high.</td></tr>
<tr><td>-input_power_pin power_pin</td><td>Identifies the input power pin.
This option is usually specified for power shifting and used with -output_power_pin.</td></tr>
</table>

Arguments	**-output_power_pin** *power_pin*	Identifies the output power pin. This option is usually specified for ground shifting and used with **–input_power_pin**.
	-input_ground_pin *ground_pin*	Identifies the input ground pin. This option is usually specified for ground shifting and used with **–output_ground_pin**.
	-output_ground_pin *ground_pin*	Identifies the output ground pin. This option is usually specified for ground shifting and used with **–input_ground_pin**.
	-ground *ground_pin*	Identifies the ground pin of the cell. This option can only be specified for level-shifters that only perform power shifting. In other words, it is an error to use this option with **–input_ground_pin** and **–output_ground_pin**.
	-power *power_pin*	Identifies the power pin of the cell. This option can only be specified for level-shifters that only perform ground shifting. In other words, it is an error to use this option with **–input_power_pin** and **–output_power_pin**.
	-enable *pin*	Identifies the pin that prevents internal floating when the power supply of the originating power domain is powered down, but the output voltage level power pin remains on. The related power and ground of this pin is the output power and ground pins defined for this cell.
	-pin_groups **{{***input_pin output_pin* [*enable_pin*]**}*}**	Specifies a list of input-output paths for multi-bit isolation cells. Each group in the list specifies one cell input pin, one cell output pin, and one optional enable pin that applies to the specified path. An enable pin may appear in more than one group. It is an error if the same input or output pin appears in more than one group.
	-valid_location <**source** | **sink** | **either** | **any**>	Specifies the valid location of the level-shifter cell. The default value is **sink**.
	-bypass_enable *expression*	Specifies when to bypass the voltage shifting functionality. When the expression evaluates to *True*, the cell behaves like a buffer. The expression shall be a simple expression of the bypass enable input pin. By default, the related power and ground of this pin is the output power and ground pin defined for this cell.
	-multi_stage *integer*	Identifies the stage of a multi-stage level-shifter to which this definition (command) applies. For a level-shifter cell with *N* stages, *N* definitions shall be specified for the same cell. Each definition needs to associate a number from `1` to *`N`* for this option. For more information, see Annex I.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_level_shifter_cell** library command identifies the library cells to use as level-shifter cells, as follows:

— If **-input_voltage_range** is specified, the **-output_voltage_range** shall also be specified.

— If **-ground_input_range** is specified, the **-ground_output_range** shall also be specified.

— It is an error if neither **-input_voltage_range** nor **-ground_input_voltage_range** is specified.

If a list of voltages ranges is specified for the input supply voltage, a list of voltages ranges for the output supply voltage with the same number of elements shall also be specified., i.e., each member in the list of input voltage ranges needs to have a corresponding member in the list of output voltage ranges.

By default, the enable and output pins of this cell are related to the output power and output ground pins (specified through the **-output_power_pin** and **-output_ground_pin** options). And the non-enable input pin is related to the input power and input ground pins (specified through the **-input_power_pin** and **-input_ground_pin** options).

The **-valid_location** option specifies the valid location of the level-shifter cell, as follows:

a) **source**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the driving supply set for a net requiring level-shifting.

b) **sink**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the receiving supply set for a net requiring level-shifting.

c) **either**—indicates the cell shall be inserted in a location where the primary supply set is equivalent to the driving supply set or the receiving supply set for a net requiring level-shifting.

d) **any**—indicates the cell can be placed in any location.

   1) If the cell contains pins for rail connection, these pins shall not be specified through the **-input_power_pin**, **-output_power_pin**, **-input_ground_pin**, or **-output_ground_pin** options.

   2) A power level-shifter with this setting can be placed in any location as long as its primary ground net is equivalent to the driving and receiving primary ground net of the net requiring level-shifting.

   3) A ground level-shifter with this setting can be placed in any location as long as its primary power net is equivalent to the driving and receiving primary power net of the net requiring level-shifting.

   4) For a power and ground level-shifter, which requires two definitions of the command—one for the power part and one for the ground part of the cell—the **-valid_location** can be different in the two definitions.

Power part	**Ground part**
`any`	`source|sink|either`
`source|sink|either`	`any`
`any`	`any`

      i) In the first case, the ground-shifting part of the level-shifter definition determines the location.

      ii) In the second case, the power-shifting part of the level-shifter definition determines the location.

      iii) In the third case, the cell can be placed in a domain whose power and ground supplies are neither driving the logic power and ground supplies nor receiving the logic power and ground supplies.

Examples

The following example identifies level-shifter cells with one power pin and one ground pin that perform power shifting from 1.0 V to 0.8 V.

```
define_level_shifter_cell
-cells LSHL
-input_voltage_range {{1.0 1.0}} -output_voltage_range {{0.8 0.8}}
-direction high_to_low
-input_power_pin VH -ground G
```

The following example identifies level-shifter cells that perform power shifting from 0.8 V to 1.0 V. In this case, the level-shifter cells need to have two power pins and one ground pin.

```
define_level_shifter_cell
-cells LSLH
-input_voltage_range {{0.8 0.8}} -output_voltage_range {{1.0 1.0}}
-direction low_to_high
-input_power_pin VL -output_power_pin VH -ground G
```

The following example identifies level-shifter cells with valid location `any` to perform voltage shifting from 0.8 V to 1.0 V. The cells have three power pins and one ground pin.

`VDD`—This is the standard cell rail; this pin is not used by the cell.

`VDDL`—This is the power pin to which the input signal is related.

`VDDH`—This is the power pin to which the output signal is related.

`VSS`—This is the ground pin of the cell.

```
define_level_shifter_cell
-cells LSLH
-direction low_to_high
-input_voltage_range {{0.8 0.8}} -output_voltage_range {{1.0 1.0}}
-input_power_pin VDDL -output_power_pin VDDH -ground VSS
-valid_location any
```

The following example identifies level-shifter cells that perform both power shifting from 0.8 V to 1.0 V and ground shifting from 0.2 V to 0 V. In this case, the level-shifter cells need to have two power pins and two ground pins. In addition, since the input voltage swing is 0.6 V (0.8 V – 0.2 V), which is smaller than the output voltage swing of 1.0 V (1.0 V – 0 V), the direction of the cell is `low_to_high`.

```
define_level_shifter_cell
-cells LSLH
-input_voltage_range {{0.8 0.8}} -output_voltage_range {{1.0 1.0}}
-ground_input_voltage_range {{0.2 0.2}} -ground_output_voltage_range {{0.0
 0.0}}
-direction low_to_high
-input_power_pin VL -output_power_pin VH
-input_ground_pin GH -output_ground_pin GL
```

The following example indicates the level-shifter can shift from 0.8 V to 1.0 V or from 1.0 V to 1.2 V. However, the cell cannot shift power voltage from 0.8 V to 1.2 V.

```
define_level_shifter_cell
-cells LSLH
-input_voltage_range {{0.8 1.0}} -output_voltage_range {{1.0 1.2}}
-input_power_pin VL -output_power_pin VH -ground_pin VSS
-direction low_to_high
```

The following example indicates the level-shifter can shift from input range 0.8 V to 0.9 V to output range 1.0 V to 1.1 V, or from input range 0.9 V to 1.0 V to output range 1.1 V to 1.2 V. Note that the cell cannot shift input voltages between 0.8 V to 0.9 V to output voltages 1.1 V to 1.2 V.

```
define_level_shifter_cell
-cells LSLH -input_power_pin VL -output_power_pin VH -ground_pin VSS
-input_voltage_range {{0.8 0.9} {0.9 1.0}}
-output_voltage_range {{1.0 1.1} {1.1 1.2}}
-direction low_to_high
```

The following examples illustrate the use of the **-pin_groups** option to specify multi-bit level-shifter cells with and without `enable`:

```
define_level_shifter_cell -cells mbit_en_ls -pin_groups { { datain1
   els_dataout1 en1 } {datain2 els_dataout2 en2 } }
define_level_shifter_cell -cells mbit_ls -pin_groups { { datain1
   ls_dataout1 } { datain2 ls_dataout2 } }
```

7.6 define_power_switch_cell

Purpose	Identify a power switch or ground-switch cell	
Syntax	**define_power_switch_cell** **-cells** *cell_list* **-type** <**footer** | **header**> **-stage_1_enable** *expression* [**-stage_1_output** *expression*] {**-power_switchable** *power_pin* **-power** *power_pin* | **-ground_switchable** *ground_pin* **-ground** *ground_pin*} [**-stage_2_enable** *expression* [**-stage_2_output** *expression*]] [**-always_on_pins** *ordered_pin_list*] [**-gate_bias_pin** *power_pin*]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as power-switch cells.
	-type <**footer** | **header**>	Specifies whether the power-switch cell is a **header** or **footer** cell.
	-stage_1_enable (**-stage_2_enable**) *expression*	Specifies when the switch cell driven by this input pin is turned on (enabled) or off. If only stage 1 is specified, the switch is turned on when the expression for the **-stage_1_enable** option evaluates to *True* and the switch is turned off when the expression for the **-stage_1_enable** option evaluates to *False*. If both stages are specified, the switch is turned on when the expression for both enable options evaluates to *True* and the switch is turned off when the expression for both enable options evaluates to *False*. The Boolean expression is a simple expression of the input pin.
	-stage_1_output (**-stage_2_output**) *expression*	Specifies whether the output pin in the expression is the buffered or inverted output of the input pin specified through the corresponding **-stage_*x*_enable** option. In a design, this pin is used to connect another switch cell in series to form a power-switch chain.
	-power_switchable *power_pin*	Identifies the output power pin in the corresponding cell. This option can only be used if the power gating cell is used to cut off the path from power to ground on the power side (i.e., for a header cell). This pin shall be connected to a switchable power net.
	-power *power_pin*	Identifies the input power pin of the cell.
	-ground_switchable *ground_pin*	Identifies the output ground pin in the corresponding cell. This option can only be used if the power gating cell is used to cut off the path from power to ground on the ground side (i.e., for a footer cell). This pin shall be connected to a switchable ground net.

Arguments	**-ground** *power_pin*	Identifies the input ground pin of the cell.
	-always_on_pins *ordered_pin_list*	Specifies a list of cell pins related to the input power and ground pins of the cell.
	-gate_bias_pin *power_pin*]	Identifies a power pin that provides the supply used to drive the gate input of the switch cell.
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_power_switch_cell** library command identifies the library cells to use as power-switch cells. The input enable and output enable pins of these cells are related to the non-switchable power and ground pins.

Examples

The following example defines a header power switch. The power switch has two stages. The power switch is completely on if the transistors of both stages are on. The stage 1 transistor is turned on by applying a low value to input `I1`. The output of the stage 1 transistor, `O1`, is a buffered output of input `I1`. The stage 2 transistor is turned on by applying a high value to input `I2`. The output of stage 2 transistor, `O2`, is the inverted value of input `I2`.

```
define_power_switch_cell -cells 2stage_switch -stage_1_enable !I1
-stage_1_output O1 -stage_2_enable I2 -stage_2_output !O2 -type header
```

7.7 define_retention_cell

Purpose	Identify state retention cells	
Syntax	**define_retention_cell** **-cells** *cell_list* **-power** *power_pin* **-ground** *ground_pin* [**-cell_type** *string*] [**-always_on_pins** *pin_list*] [**-restore_function** {{**pin** <**high** | **low** | **posedge** | **negedge**}}] [**-save_function** {{**pin** <**high** | **low** | **posedge** | **negedge**}}] [**-restore_check** *expression*] [**-save_check** *expression*] [**-retention_check** *expression*] [**-hold_check** *pin_list*] [**-always_on_components** *component_list*] [**-power_switchable** *power_pin*] [**-ground_switchable** *ground_pin*]	
Arguments	**-cells** *cell_list*	Identifies the specified cells as state retention cells.
	-power *power_pin*	Identifies the power pin of the cell. If this option is specified with the **-power_switchable** option, it indicates this is a non-switchable power pin.
	-ground *ground_pin*	Identifies the ground pin of the cell. If this option is specified with the **-ground_switchable** option, it indicates this is a non-switchable ground pin.
	-cell_type *string*	Specifies a user-defined name grouping the specified cells into a class of retention cells that all have the same retention behavior. This specification limits the group of cells that can be used to those requested through the **–lib_cell_type** option of the **map_retention_cell** command (see 6.33).
	-always_on_pins *pin_list*	Specifies a list of cell pins that are related to the nonswitchable power and ground pins of the cells.
	-restore_function {{**pin** <**high** | **low** | **posedge** | **negedge**}}	Specifies the polarity or the edge sensitivity of the restore pin that enables the retention cell to restore the saved value after exiting power shut-off mode. By default, the restore pin relates to the non-switchable power and ground pin of the cell. If not specified, the restore event is triggered when the primary power is restored, or the power-up event. When neither **-save_function** nor **-restore_function** is specified, the current value is always saved before entering retention mode and the saved value is restored when the primary power is restored.
	-save_function {{**pin** <**high** | **low** | **posedge** | **negedge**}}	Specifies the polarity or the edge sensitivity of the save pin that enables the retention cell to save the current value before entering retention mode. By default, the save pin relates to the non-switchable power and ground pin of the cell. If not specified, the save event is triggered by the negation of the restore function when it is specified. When neither **-save_function** nor **-restore_function** is specified, the current value is always saved before entering retention mode and the saved value is restored when the primary power is restored.
	-restore_check *expression*	Specifies the additional condition when the states of the sequential elements can be restored. The expression shall be a function of the cell input pins. The expression shall be *True* when the restore event occurs.

Arguments	**-save_check** *expression*	Specifies the additional condition when the states of the sequential elements can be saved. The expression shall be a function of the cell input pins. The expression shall be *True* when the save event occurs.
	-retention_check *expression*	Specifies an additional condition to meet (after the primary supply of the retention cell is switched off and before the supply is powered on again) for the retention operation to be successful. *expression* can be a Boolean function of cell input pins. The *expression* shall be *True* when the primary supply set of the power domain, in which the retention logic locates, is shut off and the retention supply set is on.
	-hold_check *pin_list*	Specifies a list of pins that maintain the same logic value during the retention period, from the time when the save event occurs to the time when the restore event occurs. The pin may be the clock pin or any other control pin.
	-always_on_components *component_list*	Specifies a list of component names: instances, named processes, sequential reg, or signal names, in the corresponding simulation model that are powered by the nonswitchable power and ground pins. The logic values of the specified components are corrupted if the state value of the nonswitchable power and group pin is `OFF`. NOTE—The option has only an impact on tools that use the gate-level simulation models of state retention cells.
	-power_switchable *power_pin*	Identifies the switchable ground pin. This cell can be used for retention purpose in a power domain that can be shutoff using power switches (i.e., using a header cell).
	-ground_switchable *ground_pin*	Identifies the switchable ground pin. This cell can be used for retention purpose in a power domain that can be shutoff using ground switches (i.e., using a footer cell).
Return value	Return an empty string if successful or raise a `TCL_ERROR` if not.	

The **define_retention_cell** library command identifies the library cells to use as retention cells. The following also apply:

— By default, all pins of this cell are related to the switchable power and ground pins, unless otherwise specified.

— It is an error if the save and restore functions both identify the same pin, and the polarity or edge sensitivity are the same for that pin. For example, the following two commands are incorrect:

```
define_retention_cell -cells My_Ret_Cell1
  -restore_function {pg high} -save_function {pg high}
define_retention_cell -cells My_Ret_Cell2
  -restore_function {pg posedge} -save_function {pg posedge}
```

— It is an error if the conditions specified in **-save_check**, **-restore_check**, or **-retention_check** conflict with **-hold_check**. For example, the specification

```
'-hold_check clk -save_check !clk -restore_check clk'
```

is an error since the `-hold_check` requires the `clk` signal to hold the same value from the time when the save event occurs to the time when the restore event occurs, but the other two options require the signal `clk` have different values.

NOTE—If the cell data output pin is listed in the **-always_on_pins** list, then this retention cell may be used for retention strategies that specify **-use_retention_as_primary**.

Example

In the following example, the cell design requires clock `clk` be held to `0` to save or restore the state of the sequential element. If retention control pin `save` is set to `0`, the state will be saved and saved data will be restored when the primary power `VDD` is restored. The retention power `VDDC` shall be on to enable the retention while `VDD` is switched off.

```
define_retention_cell -cells My_Ret_Cell -power VDDC
-ground VSS -power_switchable VDD
-save_check {!clk} -restore_check {!clk}
-save_function {save negedge}
```

8. UPF processing

8.1 Overview

All UPF commands have an immediate effect when they are executed by a Tcl interpreter. For the following commands, the immediate effect is the only effect:

- **create_hdl2upf_vct** (see 6.14)
- **create_upf2hdl_vct** (see 6.23)
- **find_objects** (see 6.26)
- **load_simstate_behavior** (see 6.27)
- **load_upf** (see 6.28)
- **load_upf_protected** (see 6.29)
- **set_design_top** (see 6.38)
- **set_design_attributes** (see 6.37)
- **set_port_attributes** (see 6.46)
- **set_scope** (see 6.52)
- **set_simstate_behavior** (see 6.53)
- **set_partial_on_translation** (see 6.44)
- **upf_version** (see 6.54)

All other UPF commands have both an immediate and a deferred effect. For these commands, the immediate effect is to add the command syntax to an internal structure for further processing. The deferred effect varies with the command, but typically contributes to construction of a power intent model reflecting the specification. This model is then applied to the design as appropriate for the tool involved.

One exception is the **save_upf** command (see 6.36), for which the deferred effect is generation of a UPF file describing the power intent for a given scope. This generation occurs after the power intent model has been fully constructed, so the generated UPF file is complete.

NOTE—This algorithm defines a reference model for UPF command processing, to illustrate how the interdependencies between design data and the UPF specification, and among UPF commands themselves, can be satisfied. A given tool may use a different algorithm as long as the overall effect is the same as this algorithm would present.

8.2 Data requirements

In addition to the UPF file(s) involved, UPF processing requires access to the following data:

- Elaborated design hierarchy
- UPF attribute specifications in HDL (if any)
- Library cell definitions

These data need to be available when UPF processing begins.

8.3 Processing phases

The following describes the detailed sequence of operations to process a UPF description, extract the power intent it specifies, and apply the power intent to a design for use in a verification or implementation tool.

8.3.1 Phase 1—read and resolve UPF specification

In this phase, the UPF commands are parsed and further processed to create a normalized representation of the UPF specification. This involves the following operations:

a) Read and execute each UPF command as it is read in
   1) Resolve references to the design relative to the current scope
   2) Execute **create_logic_port** (see 6.16), **create_logic_net** (see 6.15), and **connect_logic_net** (see 6.10)
   3) Execute **find_objects** commands (see 6.26) on elaborated design (unmodified)
   4) Build/extend syntactic model of UPF specification

b) Augment syntactic model of UPF specification with HDL-specified and library-specified UPF attributes

c) Collapse **-update** commands and check for conflicts

d) Apply defaults for defaultable options

In general, names shall be defined before being referenced. In this phase, name-defining UPF commands are associated with the scope in which the object is defined, or with the parent object for which a subordinate object is defined, as appropriate, so that subsequent name references can be resolved at this stage.

Names of design objects referenced in UPF commands shall be defined in the design hierarchy before they are referenced in UPF. Names of the library cells referenced in UPF commands shall be defined for the design before they are referenced in UPF. Names of UPF-defined objects shall be defined and associated with the appropriate design hierarchy scope before they are referenced in UPF. Names of objects that are associated with other objects (supply set handles of power domains; functions of supply sets or supply set handles; port states of ports; power states of supply sets, power domains, or modules; simstates of power states) shall be defined and associated with the relevant parent object before they are referenced in UPF. Names of VCTs shall be defined in UPF and associated with the global VCT scope before they are referenced in UPF.

Any command that updates a previous command that defined a simple name in a design hierarchy scope shall be processed in the scope in which the original command was processed and be associated with that same scope. Any command that updates a previous command that defined an object associated with a parent object shall also be processed in the scope in which the original command was processed and be associated with that same parent object.

8.3.2 Phase 2—build power intent model

In this phase, the normalized UPF specification is executed to construct a model of the power intent expressed by the specification. This involves the following operations:

a) Construct power domains
   1) As specified by **create_power_domain** commands (see 6.17)
   2) Using the effective element list algorithm in 5.10
   3) Including constructing required supply sets and functions
   4) Atomic power domains shall be constructed first, followed by non-atomic power domains

b) Construct control logic for isolation, retention, and switch instances

   As specified by **create_logic_*** (see 6.15 and 6.16) and **connect_logic_net** (see 6.10) commands

c) Construct supply networks and connections to power domains/strategies
   1) As specified by **create_supply_*** (see 6.20, 6.21, and 6.22) and **create_power_switch** (see 6.18) commands

2) **connect_supply_*** (see 6.11 and 6.12), **create_*_vct** (see 6.14 and 6.23), and **associate_supply_set** (see 6.7) commands
   i) Including equivalent supply declarations
   ii) Including error checks related to supply set/function association

d) Construct explicit, implicit, and automatic supply connections

   As specified by **connect_supply_*** commands (see 6.11 and 6.12), **associate_supply_set** (see 6.7), etc.

e) Apply the power model of a hard IP cell as specified by **apply_power_model** command (see 6.6)

f) Construct composite domains
   1) As specified by **create_composite_domain** (see 6.13) commands
   2) Including propagation of primary supply to/among subdomains
   3) Including error checks related to domain composition

g) Identify power-domain boundary ports and their supplies

   By analyzing the elaborated design and **create_power_domain** (see 6.17) commands

h) Apply retention strategies for each domain

   As specified by **set_retention** (see 6.49 and 4.5.6)

i) Apply repeater strategies for each domain

   As specified by **set_repeater** (see 6.48 and 4.5.6)

j) Apply isolation strategies for each domain boundary port

   As specified by **set_isolation** (see 6.41 and 4.5.6)

k) Apply level-shifting strategies for each domain boundary port

   As specified by **set_level_shifter** (see 6.43 and 4.5.6)

l) Identify cells to use for isolation, level-shifting, retention, and switch elements

   As specified by **map_*** (see 6.32 and 6.33) and **use_interface_cell** (see 6.55) commands

m) Construct power states

   As specified by **add_power_state** (see 6.4) commands

n) Construct power state transitions

   As specified by **describe_state_transition** (see 6.24) commands

8.3.3 Phase 3—recognize implemented power intent

In this phase, the **-instance** options of all commands are processed to identify instances of cells that implement the power intent. If a given command has a **-instance** option, this indicates that the command has been implemented by some preceding step in the flow. The implementation may or may not be complete. In particular, new logic added to the design by some tool step (e.g., for test insertion) may trigger further implementation through another application of the same command.

If a given command has a **-instance** option that specifies an empty string as the instance name, this indicates the instance resulting from applying the command in this particular context has been optimized away. In this case, tools shall not infer a cell for this application of the command. In particular, verification tools shall not infer a cell for purposes of verification, and implementation tools shall not re-implement the command by inserting a cell again.

If a given command has a **-instance** option that specifies a hierarchical name as the instance name, the specified instance shall exist in the design. It shall be an error if that hierarchical name does not identify a cell instance of the appropriate type for the command. Attributes specified in library cells, in HDL models, or in UPF may be used to determine whether a given cell instance is appropriate for the command whose

-instance option identifies it as resulting from the implementation of that command. In this case also, tools shall not infer a cell for this application of the command. Instead, the existing cell shall be used.

In addition to the preceding, commands that create supply or logic ports or nets are processed to identify any ports or nets that already exist in the HDL hierarchy. If a **create_supply_port** (see 6.21), **create_supply_net** (see 6.20), **create_logic_port** (see 6.16), or **create_logic_net** (see 6.15) command specifies a port or net name that already exists in the current scope of the HDL hierarchy, it shall be an error if that port or net name does not identify a port or net, respectively, of the appropriate type for the command. A supply port or net is appropriate for a **create_supply_port** or **create_supply_net** command, respectively, if it is declared to be of type `supply_net_type` defined in the package UPF. A logic port or net is appropriate if it is declared with the standard logic type in the relevant HDL. In this case also, tools shall not create a new port or net for this application of the command. Instead, the existing port or net shall be used.

8.3.4 Phase 4—apply power intent model to design

In this phase, some or all of the power intent model is applied to the HDL design. A given tool will add the power intent elements required for that tool's operation to the design model. Power intent model elements that are already present in the design will not be added again. This includes implementation of any checkers introduced by the **bind_checker** command (see 6.9).

NOTE—It may be appropriate for a given tool to update existing elements in the design to more completely reflect the power intent model. For example, a tool may choose to change the data type of a net in the design used as a supply net, from a single-bit type to the appropriate (SystemVerilog or VHDL) `supply_net_type`.

8.4 Error checking

Error checking is done in various UPF processing stages. Error checks include the following classes of checks, which would be performed in Phases 1, 2, and 3 of UPF processing:

a) Phase —Read and resolve UPF specification (see 8.3.1)

   1) UPF syntax checks (including semantic restrictions)

   2) Update conflict checks

   3) Design scope/object reference checks (scope/object not found)

b) Phase 2—Build power intent model (see 8.3.2)

   1) Conflicts between two commands applying to same object

   2) Completeness checks (e.g., all instances are in a power domain)

c) Phase 3—Identify implemented power intent (see 8.3.3)

   Name conflicts (an existing design object conflicts with a UPF name)

If a tool detects and reports an error in any of the preceding UPF processing phases, the tool may continue processing if possible, in order to identify any additional errors that might exist in the UPF specification or its interpretation with the design hierarchy, but processing should terminate before Phase 4, where the power intent model is applied to the design hierarchy.

9. Simulation semantics

This clause details the simulation semantics for the UPF commands (see also Clause 6).

9.1 Supply network creation

UPF supply network creation commands define the power supply network that connects power supplies to the instances in a design. After these commands are applied, every instance in a design is connected to the power supply network. The *supply network* is a set of supply nets, supply ports, switches, and potentially, regulators and generators. Supply sets are defined in terms of supply nets and conveniently define a complete power circuit for instances. Supply sets simplify the management of related supply nets and facilitate connections based on the role the supply set provides for a power domain and the functions the supply nets provide within the set (see 9.2.2). The supply network defines how power sources are distributed to the instances and how that distribution is controlled.

A supply port that propagates but does not originate a supply state and voltage value defines a *supply source*. At any given time, a supply source can be traced through the supply network connectivity to a single root supply driver. The output port of a switch is a root supply source (with a corresponding driver); the value of its driver is computed according to the algorithm given in the following item h). HDL switch models should use the `assign_supply2supply` function to propagate the input supply to the output supply. `assign_supply2supply` propagates or maintains the trace back of the root supply driver information. Bias generators, voltage regulators, and switches modeled in HDL should create a root supply driver when the supply source originates from within the model.

Determination of the root supply driver is required for certain supply network resolution functions (see 6.20).

NOTE—Since the supply net type is defined in the package UPF, it is possible to create the supply network entirely in HDL source.

A supply net can be connected to a port declared in the HDL description. In this case, the supply net state is connected to the port; the voltage is not used. VCTs define the conversion from supply net state values to values of an HDL type and vice versa to facilitate more complex modeling consistent with an organization's logic value interpretations of UPF supply port states.

If a supply net is connected to a HDL port of a single bit type, a default VCT that maps the **FULL_ON** state to logic 1 and the **FULL_OFF** state to logic 0 shall be inserted automatically. The default VCT facilitates building simple functional models. If this mapping is not the one desired for a particular connection, a user-defined VCT implementing the desired mapping can be specified explicitly for the connection (see also Annex F).

Supply port/net interconnections create a supply network that may span multiple instances at potentially multiple levels in the logic hierarchy. Evaluation of supply networks during simulation requires consideration of the whole collection of electrically equivalent supply ports/nets (see 4.4.3) making up each supply network.

a) A group of electrically equivalent ports/nets (see 4.4.3) constitutes a supply network, including ports/nets that are both equivalent by connection and declared electrically equivalent.
    1) The source(s) of the group are the top-level and leaf-level sources.
    2) The load(s) of the group are the top-level and leaf-level loads.
    3) Internal ports act only as connections within the group.
b) If there are no resolved nets in the group, then the group is unresolved.
c) For an unresolved group, it is an error if there is more than one supply source in the group.

d) If there is at least one resolved net in the group, then the group is resolved.

e) For a resolved group, it is an error if
   1) the group contains two resolved nets with different resolution types;
   2) any two resolved nets in the group are separated by a unidirectional internal port.

f) In general, it is an error if a unidirectional supply port (an input port or an output port) in the group
   1) has a supply source on the load side, and
   2) has a load on the supply source side.

g) For an unresolved group of electrically equivalent supply ports/nets (see 4.4.3), the single source drives all the loads directly.

h) For a resolved group of electrically equivalent supply ports/nets
   1) all electrically equivalent resolved nets in a group are collapsed into a single resolved net;
   2) supply sources provide inputs to the resolved net;
   3) the resolution type of the resolved net determines how inputs are resolved;
   4) the resolved value is distributed to all loads.

9.2 Supply network simulation

9.2.1 Supply network initialization

Simulation initialization semantics are defined by each HDL. Existing models rely on the HDL initialization semantics for operations such as initializing ROMs, etc. To ensure that initialization of the design occurs correctly during power-aware simulation, model initialization code and design code should be cleanly separated. In Verilog or SystemVerilog, initial blocks can be used for model initialization code, since these are not affected by power-aware simulation semantics. In VHDL, model initialization code should be placed in processes that will not be synthesized and these processes should be included in an “always-on” power domain during power-aware simulation.

The initial state of supply ports and supply nets is **OFF** with an unspecified voltage value. The initial state of a supply set is determined by the initial state of each supply function of the supply set. The initial state of a supply set function is determined by the initial state of the corresponding supply net with which it has been associated or else the initial state of the root supply driver of that function.

NOTE—Implicitly created supply nets are initialized the same as explicitly created supply nets.

To facilitate modeling of non-inferable behavior in HDLs that can be used in both a UPF simulation and a traditional non-UPF simulation, the following are provided:

— Predefined constant of Boolean type: **UPF_POWER_AWARE**.

  The value of this constant is **TRUE** in a UPF simulation, otherwise it is **FALSE**. This constant value is globally static only in a UPF simulation; i.e., its value is known at the time that SystemVerilog and VHDL `generate` statements are evaluated allowing the ability to specify logic that is conditionally generated only in a UPF simulation.

— In VHDL, a signal and, in SystemVerilog, a variable of type `power_state_simstate` can be declared within an architecture or module.

  The name of this signal/variable shall be `upf_simstate`. `upf_simstate` can be used in a process’s sensitivity list. It shall be an error if `upf_simstate` is assigned or connected to a port—it can only be used locally and in a read-only context. In a UPF simulation, `upf_simstate` shall represent the active simstate of the supply set that is implicitly, automatically, or explicitly connected to the instance when simstate behavior has been enabled for that element. If simstate behavior is disabled for the element, then `upf_simstate` shall remain the constant value **CORRUPT**.

9.2.2 Power-switch evaluation

During simulation, a power switch created with **create_power_switch** corresponds to a process that is sensitive to changes in its input port (net state and voltage value), as well as its control ports. [A general introduction to power-switch behavior is described here (see 6.18 for the complete power-switch semantics).] Whenever the signals on the control ports change, the corresponding on-state Boolean functions are evaluated. If an on-state function evaluates *True*, the switch is closed, which causes the state of its input port to propagate to the output port (or for a multiplexed switch, the corresponding input is switched to the output), otherwise the switch is opened—the output supply port is assigned the state **OFF** and the voltage value is unspecified. If any of the control signals is `X` or `Z`, the input supply port is **UNDETERMINED**, the control signals match one of the error-state Boolean functions, or more than one on-state function evaluates *True*, then the behavior of the output supply port is assigned the state **UNDETERMINED**, the voltage level shall be unspecified, and the acknowledge ports shall be driven `X`; in this case, implementations may issue a warning or an error.

Example

Using the following **create_power_switch** command (see 6.18):

```
create_power_switch kb
-output_supply_port {outp pda_vdd}
-input_supply_port {inp1 yt}
-input_supply_port {inp2 db}
-control_port {cp1 eh}
-control_port (cp2 as}
-on_state {yt_on_kb inp1 {(cp1 && !cp2)}}
-on_state {db_on_kb inp2 {(!cp1 && cp2)}}
-ack_port {ap yack 1}
```

creates an instance of an anonymous switch model that is functionally equivalent to the following SystemVerilog module definition:

```
import UPF::*;
module <anon> (
  output supply_net_type outp,
  output logic ap,
  input supply_net_type inp1, inp2,
input logic cp1, cp2 );

upf_object_handle in1H, in2H, outH;

initial begin
in1H = get_object( "inp1" );
in2H = get_object( "inp2" );
outH = get_object( "outp" );
if (!is_valid_handle( in1H ) || !is_supply_kind( in1H ) ||
   !is_valid_handle( in2H ) || !is_supply_kind( in2H ) ||
   !is_valid_handle( outH ) || !is_supply_kind( outH ))
 $display( "Invalid supply port connection on switch port" );
end

always@(cp1, cp2, inp1, inp2)
  case ({cp1, cp2})
    01 : begin
           assign_supply2supply( outp, inp2  );
           ap <= 1;
         end
```

```
      10 : begin
             assign_supply2supply( outp, inp1 );
             ap <= 1;
           end
      00 :
      11 :
           begin
             assign_supply_state( outp, OFF );
             ap <= 0;
           end
      default : begin

                  assign_supply_state( outp, UNDETERMINED );
                  ap <= X;

                  $stop
                end
endmodule
```

The instance of the anon module is:

```
<anon> kb (.outp(pda_vdd), .inp1(yt), .inp2(db), .ap(yack), .cp1(eh),
.cp2(as));
```

9.2.3 Supply network evaluation

During simulation, each supply port and net maintains two pieces of information: a supply state and a voltage value. The supply state itself consists of two pieces of information: an on/off state and a full/partial state. The supply state values are **FULL_ON**, **OFF**, **PARTIAL_ON**, and **UNDETERMINED**. **PARTIAL_ON** typically represents a resolved supply net state when some, but not all, switches are **FULL_ON** or any switch is **PARTIAL_ON** (see also 6.20.2).

During simulation, the supply network is evaluated repeatedly whenever the value of a root supply driver or a switch input changes. Supply network evaluation consists of the following:

a) Evaluation and resolution of supply nets (see 6.20.2)
b) Evaluation of power switches (see 6.18)
c) Evaluation of supply set power states (see 9.3)
d) Evaluation and application of simstates (see 9.4 and 9.5).

The supply network is evaluated in the same step of the simulation cycle as the logic network. New root supply driver values are propagated along the connected supply nets in the same manner that logic values are propagated along the logic network.

NOTE—As no material distinction between **PARTIAL_ON** and **PARTIAL_OFF** exists, only **PARTIAL_ON** is defined.

9.3 Power state simulation

9.3.1 Power state control

The power state of a root supply set may be changed from an HDL test bench in simulation using the `set_power_state` function defined in the package UPF (see Annex B). The `set_power_state` function changes the power state of the specified supply set (or supply set handle) to one of the states defined for the supply set (handle). This function can be used to control the supply states of root supply sets, before top-level supply networks have been implemented or completed.

When set_power_state is used to change a supply set's power state to a specified power state:

a) It is an error if the specified power state is defined with either a logic expression or a supply expression.

b) It is an error if any one of the supply set functions is associated with an explicitly declared supply net, either in the declaration of the supply set or via association of a supply set with a supply set handle.

c) The implicitly created supply nets of the set (e.g., `primary.power`), shall have their state set as follows:

   1) If the simstate of the specified power state is **CORRUPT**: the state shall be set to **OFF** and the voltage value is unspecified.

   2) For any other simstate: the state shall be set to **FULL_ON** and the voltage value is unspecified.

The `set_power_state` function cannot be used to set the power state of a power domain. However, setting the power state of a supply set or supply set handle to a given power state may indirectly affect the power state of a power domain, just as would occur if the power state of the supply set or supply set handle changed to the given power state as a result of the state of the supply network driving the root supply sets.

NOTE—Tools may provide other mechanisms to change the power state of the supply set or power domain. Such mechanisms are outside the scope of this standard.

9.3.2 Power state determination

Each supply set and each power domain may have an associated set of named power states. Each named power state is defined in terms of the values of supply ports or nets, or the power states of other supply sets or power domains, or logic signals representing control conditions, or some combination thereof.

A supply set or power domain is in a given power state *S* at a given time *T* if the definition of *S* is satisfied at time *T* by the current values of any supply or logic ports or nets referenced in the definition and by the current power states of any supply sets or power domains referenced in the definition. More than one power state definition can be satisfied at the same time, so a supply set or power domain may be in multiple power states at any given time.

The power state of a supply set is determined after all signals (including supply nets; see 9.2.3) have been updated and prior to the evaluation of the power state(s) of power domains. The power state of a power domain is determined after the power state(s) of all supply sets have been determined and prior to evaluation of user-defined processes and always blocks.

The power state of a supply set (or supply set handle) is evaluated whenever there is

a) a change in the value of any supply set (handle) function, supply net, or logic net referenced in any power state definition of the supply set, or

b) a call to the `set_power_state` function for this supply set.

The power state of a supply set is determined as follows:

```
for a supply set SS
  power state set CPS = {}
  for each power state PS defined for SS
      if PS has neither a supply expression nor a logic expression, then
         if set_power_state was called to set the power state to PS, then
            CPS = CPS + {PS}
         end
     else if PS has a supply expression but no logic expression, then
        if the supply expression is True, then
           CPS = CPS + {PS}
```

```
        end
    else if PS has a logic expression but no supply expression, then
        if the logic expression is True, then
           CPS = CPS + {PS}
        end
    else (PS has both a logic expression and a supply expression)
         if the logic expression is True, then
           CPS = CPS + {PS}
           if the supply expression is False, then
              Error: Supply status insufficient to support power state
           end
        end
     end
  end
  if CPS = {}, then
     CPS = CPS + {DEFAULT_CORRUPT}
  end
  current power states of SS = CPS
end
```

The power state of a power domain is evaluated whenever there is

c) a change in the set of current power states of any supply set (handle) or other power domain referenced in any power state definition of the power domain, or

d) a change in the value of any supply net or logic net referenced in any power state definition of the power domain.

The power state of a power domain is determined as follows:

```
for a power domain PD
   power state set CPS = {} #empty set
   for each power state PS defined for PD
      if PS has a logic expression, then
         if the logic expression is True, then
            CPS = CPS + {PS}
         end
      end
   end
   current power states of PD = CPS
end
```

9.4 Simstate simulation

The current simstate of a supply set (or supply set handle) is reevaluated whenever there is a change in the set of current power states of the supply set. If no power state in the set defines a simstate, then the current simstate remains unchanged. Otherwise, the current simstate of the supply set is set to the most corrupting simstate defined for any power state in the set of current power states of the supply set.

Each simstate has well-defined simulation semantics, as specified in the following subclauses. Multiple power states may be defined with the same simstate specification. The simstate semantics are applied to all elements that have the supply set connected to it (including no supply net connections except those implied by the supply set connection to the element) and that have the simstate semantics implicitly or explicitly enabled.

Elements implicitly connected to a particular supply set have simstate semantics enabled by default. Elements automatically or explicitly connected to a particular supply set have simstate semantics disabled by default. Use **set_simstate_behavior** to override the default enablement of simstate semantics (see 6.53).

The supply set powering a state element or the driver for a net may be in a state that the supply is not adequate to support normal operational behavior. Under specified circumstances while in these states, the logic value of the state element or net becomes unknown. A corrupt value for a state element or net indicates the logic state of the state element or net is unknown due to the state of the supply powering the state element or driver of the net. The corrupt value of a state element or net shall be the HDL's default initial value for that object's type, except for VHDL `std_ulogic` and `std_logic` typed-objects, which shall use `X` as the corruption value (not `U`).

NOTE—An object may be declared with an explicit initial value. This explicit initial value has no relationship to the corrupt value for the object. For example, in VHDL, the objects of `Integer` type have the default initial value of `Integer'Left` (`-2147483648` for a system using 32 bits to represent `Integer` types). A process variable inferring a state element may be declared to be of type `Integer` with an initial value of `0`. The corrupt value for the variable is `Integer'Left`, not `0`.

The following subclauses define the simulation semantics for simstates. These semantics are applied to the elements connected to the supply set with simstate behavior **ENABLED**.

9.4.1 NORMAL

This state is a normal, power-on functional state. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element.

9.4.2 CORRUPT

This state is a non-functional state. For example, this state can be used to represent a power-gated/power-off supply set state. In this power state, state elements powered by the supply set and the logic nets driven by elements powered by the supply set are corrupted. The element is disabled from evaluation while this state applies.

As long as the supply set remains in a **CORRUPT** simstate, no additional activity shall take place within the elements, i.e., all processes modeling the behavior of the element become inactive, regardless of their original sensitivity list. Events that were scheduled for elements supplied by the supply set before entering this simstate shall have no effect.

9.4.3 CORRUPT_ON_ACTIVITY

This state is a power-on state that is not dynamically functional. For example, this state can be used to represent a high-voltage threshold, (body-bias) state that does not have characterized (defined) switching performance. In this simstate, the logic state of the elements is maintained unless there is activity on any of the element's inputs. Upon activity on any input, then all state elements and logic nets driven by the element are corrupted.

9.4.4 CORRUPT_ON_CHANGE

This state is a power-on state that is not dynamically functional. For example, this state can be used to represent a high-voltage threshold, (body-bias) state that does not have characterized (defined) switching performance. In this simstate, the logic state of the elements is maintained unless there is a change on any of the element's outputs. Upon change of any output, then all logic nets driven by that element output are corrupted.

9.4.5 CORRUPT_STATE_ON_CHANGE

This state is a power-on state that represents a power level sufficient to power normal functionality for combinational functionality, but insufficient for powering the normal operation of a state element if the state element is written with a new value. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element, except that any change to the stored value in a state element results in the writing of a corrupt value to the state element.

9.4.6 CORRUPT_STATE_ON_ACTIVITY

This state is a power-on state that represents a power level sufficient to power normal functionality for combinational functionality but insufficient for powering the normal operation of a state element if there is any write activity on the state element. The simulator executes the design behavior of the elements consistent with the HDL or UPF specification that defines the element, except that any activity inside state elements, whether that activity would result in any state change or not, results in the writing of a corrupt value to the state element.

9.4.7 NOT_NORMAL

This is a special, placeholder state. It allows early specification of a non-operational power state while deferring the detail of whether the supply set is in the **CORRUPT**, **CORRUPT_ON_ACTIVITY**, **CORRUPT_ON_CHANGE**, **CORRUPT_STATE_ON_CHANGE**, or **CORRUPT_STATE_ON_ACTIVITY** simstate. If the supply set matches a power state specified with simstate **NOT_NORMAL**, the semantics of **CORRUPT** shall be applied, unless overridden by a tool-specific option. **NOT_NORMAL** semantics shall never be interpreted as **NORMAL**.

The functions defined in package UPF (see Annex B) that query the simstate for a state that was originally **NOT_NORMAL** shall return the simstate to be applied in simulation for that state. e.g., **CORRUPT** for the default interpretation of **NOT_NORMAL**.

The query functions (see Annex C) that query the simstate for a state having a **NOT_NORMAL** simstate shall return **NOT_NORMAL** when it was not updated with any other simstate.

NOTE 1—Using the default interpretation of **CORRUPT** for **NOT_NORMAL** provides a conservative—the broadest corruption semantics—for simulation of the design for functional verification. However, a conservative interpretation of **NOT_NORMAL** for other tools, such as power estimation tools, might be to use a bias or lowered voltage level interpretation such as **CORRUPT_ON_ACTIVITY**.

NOTE 2—As it is possible for two or more power states of a supply set to match the state of the supply set's nets and for multiple simstate specifications to apply simultaneously, the effective result is that the simstate with the broadest corruption semantics shall apply. For example, a supply set that matches power states with simstates of **CORRUPT_STATE_ON_CHANGE** and **CORRUPT_STATE_ON_ACTIVITY** shall result in the application of **CORRUPT_STATE_ON_ACTIVITY** simstate semantics being applied.

9.5 Transitioning from one simstate state to another

The following subclauses define the simulation semantics for transitions from one simstate to another. These semantics are applied to the elements connected to the supply set with simstate behavior **ENABLED**.

9.5.1 Any state transition to CORRUPT

In this case, the nets and state elements driven by the elements connected the supply set in this simstate shall be corrupted. The elements connected to this supply set are inactive as long as the supply set is in the **CORRUPT** simstate.

9.5.2 Any state transition to CORRUPT_ON_ACTIVITY

In this case, the current state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall remain enabled for activation (evaluation). Any net or state element that is actively driven after transitioning to this state shall be corrupted.

Any attempt to restore a retention register's retained value while in the **CORRUPT_ON_ACTIVITY** state shall result in corruption of the register's value.

9.5.3 Any state transition to CORRUPT_ON_CHANGE

In this case, the current state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall remain enabled for activation (evaluation).

9.5.4 Any state transition to CORRUPT_STATE_ON_CHANGE

In this case, the current state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall be enabled for activation (evaluation).

9.5.5 Any state transition to CORRUPT_STATE_ON_ACTIVITY

In this case, the current state of nets and state elements driven by the element shall remain unchanged at the transition. The processes modeling the behavior of the element shall be enabled for activation (evaluation).

9.5.6 Any state transition to NORMAL

In this case, the processes modeling the behavior of the element shall be enabled for activation (evaluation), and the combinational and level-sensitive sequential logic functionality in each process shall be re-evaluated to restore and properly propagate constant values and current input values. Edge-sensitive sequential logic functionality within the element shall not be evaluated at this transition.

9.5.7 Any state transition to NOT_NORMAL

NOT_NORMAL is simulated according to the interpretation of this placeholder simstate (see 9.4.7).

9.6 Simulation of retention

This subclause covers some of the basics of retention register operation and modeling, which are useful in describing the simulation semantics for the **set_retention** command (see 6.49). The following abbreviations are used in various figures and tables herein:

`VDD`	primary supply port of the register
`VDDRET`	retention supply port of the register
`SS`	save signal is active
`SC`	save condition
`RS`	restore signal is active

`RC` restore condition

`RTC` retention condition

9.6.1 Retention corruption summary

A retention register has the same simulation behavior as a regular register when both supplies `VDD` and `VDDRET` are `ON`, the save/restore signals are inactive and the retention condition is *False*. The main simulation difference between a non-retention register and a retention register comes when the corruption behavior is modeled during various power state transitions. The retention register is composed of at least three components (see 4.3.4), as follows:

— *Register value* is the data held in the storage element of the register. In functional mode, this value gets updated on the rising/falling edge of clock or gets set or cleared by set/reset signals, respectively.

— *Retained value* is the data in the retention element of retention register. The retention element is powered by the retention supply.

— *Output value* is the value on the output of the register.

The retained value of the retention register can be corrupted in the following ways:

a) If `VDDRET==OFF`

Corrupt if `RET_SUP_COR` is set

b) Else If `VDDRET==ON`

1) If `VDD==ON`

`(SS && SC) && (RS && RC)` (both save/restore are true) and `SAV_RES_COR` is set

2) Else If `VDD==OFF`

i) `(SS && SC)` — trying to save when domain off

ii) `(RS && RC)` — trying to restore when domain off

iii) `!RTC`

The output value of the retention register can be corrupted in the following ways:

c) If **-use_retention_as_primary** is specified

Output is corrupted whenever retained value (described above) is corrupted.

d) If **-use_retention_as_primary** is not specified

1) If `VDD==OFF`

Corrupt always

2) Else If `VDDRET==OFF`

Corrupt if `RET_SUP_COR` is set

In summary, the preceding algorithm covers all the conditions by which a retention register (i.e., retained value/output value) can be corrupted. A corrupted retention register can then be restored to a valid state by a combination of one or more of the following:

— Restore (power up) the corrupting supplies

— Deassert save/restore signals if the corruption is due to the condition when both are true simultaneously

— Deassert retention condition

— Apply reset/set and/or clock

9.6.2 Retention modeling for different retention styles

Depending on the type of retention, the controlling inputs of the retention register like the save/restore signals may or may not exist on the register boundary. Thus, it is important to understand the modeling of the different flavors of retention, namely balloon-style retention and master/slave-alive style retention (see 4.3.4).

When the **set_retention** (see 6.49) is specified with **-save_signal** and (or) **-restore_signal**, balloon-style retention semantics are applied to it. The process of saving/restoring is unique to balloon-style retention. When the **set_retention** is not specified with both **-save_signal** and **-restore_signal** and it is specified only with a **-retention_condition**, the master/slave-alive retention semantics are applied instead. In this type of retention, the restore happens during power-up, as the master/slave latch is kept on the retention supply. However, whether to be in a retention state or not may be controlled by the value of one or more ports on the retention register.

A retention register may be in one of the following states:

NORMAL—Functional/active mode, all supplies expected to be ON.

SAVE—The time snapshot where the save action occurs (for balloon-latch style registers).

RESTORE—The time snapshot where the restore action occurs (for balloon-latch style registers).

RETAIN_ON—The time snapshot where the primary supply is `ON` and the register is in retention state (`retention_condition == True`).

RETAIN_OFF—The time snapshot where the primary supply is `OFF` and the register is in retention state (`retention_condition == True`).

PARTIAL_CORRUPT—The retained value is corrupted, but the register value is not corrupted.

CORRUPT - The register value and retained value are both corrupted.

Table 7 summarizes the power state of a balloon style retention register with respect to the states of the signals.

Table 8 summarizes the power state of a master/slave alive retention register with respect to the states of the signals.

Table 9 shows the output values of the retention register depending on the state of retention register.

Table 7—Retention power state table for balloon style retention[a]

VDD	VDD RET	SS && SC	RS && RC	RTC	Retained value	Register value	Register state	Valid next states	Comments
ON	ON	FALSE	FALSE	FALSE	Previous saved data	Previous state value	NORMAL	SAVE, RESTORE	—
ON	ON	FALSE	FALSE	TRUE	Previous saved data	Previous state value	RETAIN_ON	NORMAL, RETAIN_OFF, RESTORE	—
ON	ON	FALSE	TRUE	X	Previous saved data	Retention value	RESTORE	NORMAL, RETAIN_ON	—
ON	ON	TRUE	FALSE	X	Register value	Previous state value	SAVE	RETAIN_ON, NORMAL	—
ON	ON	TRUE	TRUE	X	CORRUPT	CORRUPT	CORRUPT	NA	SAV_RES_COR is set
ON	OFF	X	X	TRUE	CORRUPT	CORRUPT	CORRUPT	NA	—
ON	OFF	X	TRUE	FALSE	CORRUPT	CORRUPT	CORRUPT	NA	RET_SUP_COR is set
ON	OFF	X	FALSE	FALSE	CORRUPT	Previous state value	PARTIAL_ CORRUPT	NORMAL	RET_SUP_COR is set
OFF	OFF	X	X	X	CORRUPT	CORRUPT	CORRUPT	NA	RET_SUP_COR is set
OFF	ON	FALSE	FALSE	FALSE	CORRUPT	CORRUPT	CORRUPT	NA	!RTC
OFF	ON	FALSE	FALSE	TRUE	Previous saved data	CORRUPT	RETAIN_OFF	RETAIN_ON	—
OFF	ON	FALSE	TRUE	X	CORRUPT	CORRUPT	CORRUPT	NA	Restore during power-down
OFF	ON	TRUE	X	X	CORRUPT	CORRUPT	CORRUPT	NA	Save during power-down

[a]The X in this table denotes a "don't-care" condition. Valid next states are non-corrupting next states.

Table 8—Retention state table for master/slave-alive retention

VDD	VDD RET	RTC	Retained/ register value	Register state	Valid next states	Comments
ON	ON	FALSE	Previous state value	NORMAL	RETAIN_ON	—
ON	ON	TRUE	Previous state value	RETAIN_ON	NORMAL, RETAIN_OFF	—
ON	OFF	TRUE	CORRUPT	CORRUPT	NA	RET_SUP_COR is set
ON	OFF	FALSE	CORRUPT	CORRUPT	NA	RET_SUP_COR is set
OFF	OFF	X	CORRUPT	CORRUPT	NA	—
OFF	ON	FALSE	CORRUPT	CORRUPT	NA	!RTC
OFF	ON	TRUE	Retention value	RETAIN_OFF	RETAIN_ON	—

Table 9—Retention output value table[a]

use_retention_as_primary	State	Register value	Output value
TRUE	NORMAL	DATA	DATA
TRUE	RETAIN-ON/RETAIN-OFF	DATA	DATA
TRUE	SAVE	DATA	DATA
TRUE	RESTORE	DATA	DATA
TRUE	CORRUPT	X	X
FALSE	NORMAL	DATA	DATA
FALSE	RETAIN-ON/RETAIN-OFF	DATA	VDD==ON?DATA:X
FALSE	SAVE	DATA	VDD==ON?DATA:X
FALSE	RESTORE	DATA	VDD==ON?DATA:X
FALSE	CORRUPT	X	X

[a]DATA in Table 9 stands for a valid data, and X stands for corrupt data.

Figure 6 describes the sequence of transitions in balloon style retention register. In this case, the state transitions are not synchronous, i.e., they are not caused due by clock transitions.

Figure 7 describes the sequence of transitions in a master/slave-alive register. In this case, the state transitions are not synchronous, i.e., they are not caused due by clock transitions.

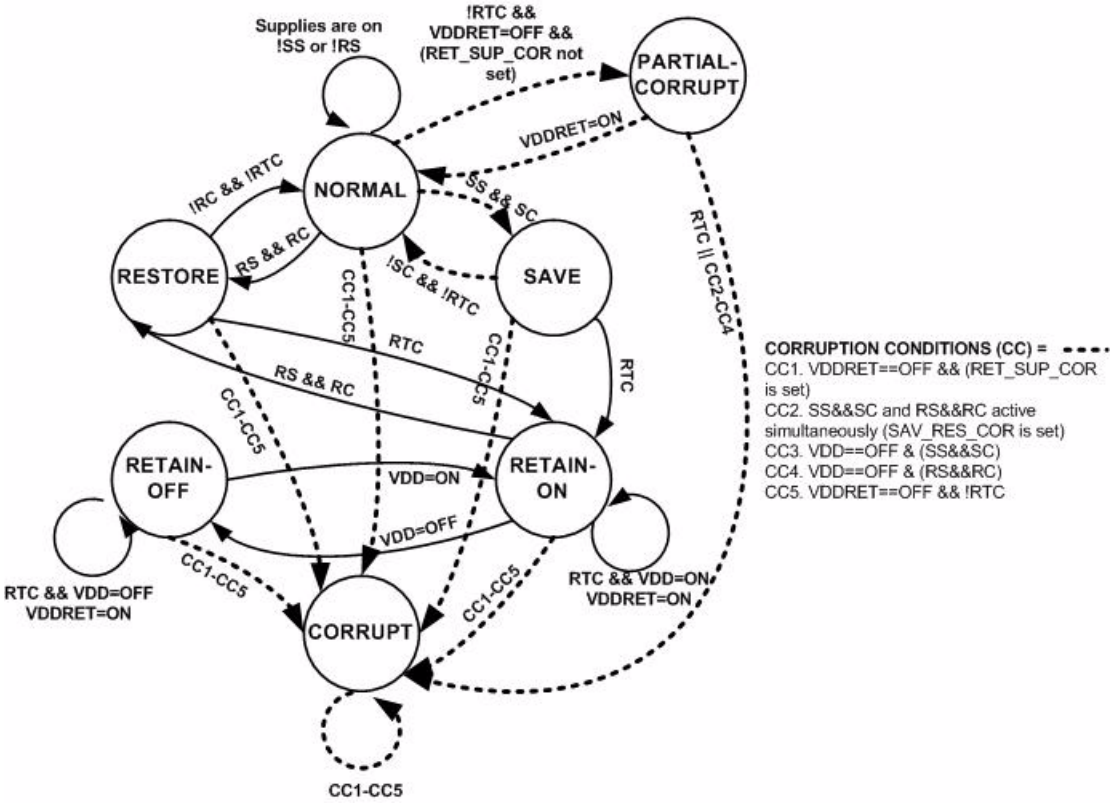

Figure 6—Retention state transition diagram for balloon-style retention

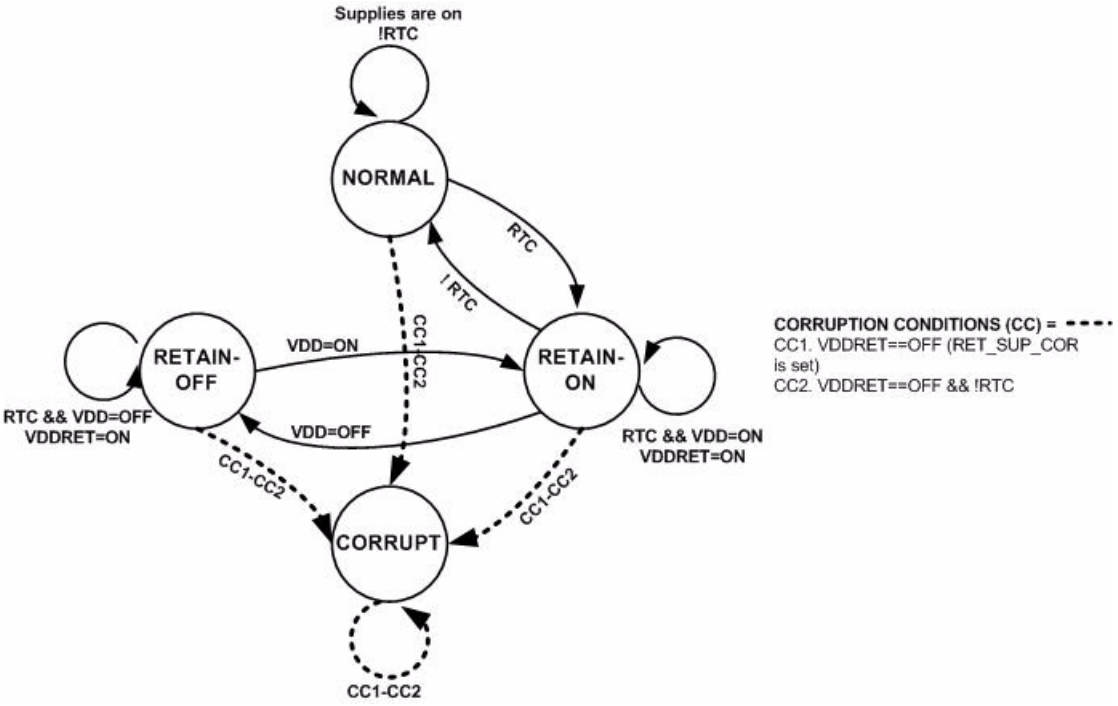

Figure 7—Retention state transition diagram for master/slave-alive style retention

9.7 Simulation of isolation

The simulation semantics for isolation are defined through an equivalent SystemVerilog `always` block, unless **-instance** applies to a specific isolation element or **use_interface_cell** (see 6.55) is applied.

An isolation strategy with a constant clamp value (`0`, `1`, `Z`, or a user-specified value) is functionally equivalent to the following SystemVerilog code:

```
// For -isolation_sense HIGH
genvar x;
generate for (x=0; x < <num_iso_specs>; x++)
always @( isolation_signal[x], <data_input>,
   <isolation_supply_set[x].simstate>)
   if (<isolation_supply_set[x].simstate> == NORMAL)
      if (isolation_signal[x] === 1'bX)
         <data_output> = <corrupt_value_for_logic_type>;
      else if (isolation_signal[x] == 1)
         <data_output> = <clamp_value[x]>;
      else
         <data_output> = <data_input>;
   else
      <data_output> = <corrupt_value_for_logic_type>;
endgenerate
```

The isolation cell with a clamp value of `latch` is functionally equivalent to the following SystemVerilog code:

```
   reg iso_latch;
   assign <isolation_output> = iso_latch;

// For -isolation_sense LOW
always @( <isolation_signal>, <non_isolated>,
          <isolation_supply_set.simstate>)
begin
   if (<isolation_supply_set.simstate> == NORMAL)
      if ( <isolation_signal === 1'bX )
         <iso_latch> = <corrupt_value_for_logic_type>;

      else if ( <isolation_signal> != 0)
        <iso_latch> = <non_isolated>;
      else
         ;
   else
      <iso_latch> = <corrupt_value_for_logic_type>;
end
```

9.8 Simulation of level-shifting

A level-shifter has the logical functionality of a buffer.

9.9 Simulation of repeater

A repeater has the logical functionality of a buffer.

Annex A

(informative)

Bibliography

Bibliographical references are resources that provide additional or helpful material but do not need to be understood or used to implement this standard. Reference to these resources is made for informational use only.

[B1] *IEEE Standards Dictionary Online*.[8]

[B2] IEEE Std 1364™, IEEE Standard for Verilog Hardware Description Language.[9, 10]

[B3] IEEE Std 1801™-2009, IEEE Standard for Design and Verification of Low Power Integrated Circuits.

[B4] ISO/IEC 8859-1, Information technology—8-bit single-byte coded graphic character sets—Part 1: Latin Alphabet No. 1.[11]

[B5] Tcl language syntax summary.[12]

[B6] Tcl language usage.[13]

[B7] Liberty library format usage.[14]

[8] *IEEE Standards Dictionary Online* subscription is available at:
http://www.ieee.org/portal/innovate/products/standard/standards_dictionary.html.

[9] IEEE publications are available from The Institute of Electrical and Electronics Engineers (http://standards.ieee.org/).

[10] The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[11] ISO/IEC publications are available from the ISO Central Secretariat (http://www.iso.org/). ISO publications are also available in the United States from the American National Standards Institute (http://www.ansi.org/).

[12] Available at http://www.tcl.tk/man/tcl8.4/TclCmd.

[13] Available at http://sourceforge.net/projects/tcl/.

[14] Available at http://opensourceliberty.org/opensourceliberty.html.

Annex G

(normative)

Supporting hard IP

When a block has an input port and an output port that are directly connected internally by a logical net, the two ports involved are called *feedthrough ports*. Tools need to recognize such ports in order to traverse through them to identify the true source(s) and sink(s) of a net. For a hard IP, automatically recognizing such ports may be difficult. To explicitly identify feedthrough ports of a hard IP, use the **feedthrough** option in a **set_port_attributes** command (see 6.46).

When a hard IP has input ports and/or output ports that are not connected internally, such ports need not be considered for any power intent specification. In addition, when performing analysis on the need of isolation or level-shifter logic at the interface of the hard IP, these ports shall be ignored. To model such ports, use the **unconnected** option in a **set_port_attributes** command (see 6.46).

G.1 Attributing feedthrough ports of hard IP

In this case, the port list shall specify the ports of a model that are all connected electrically by the same metal wire. If the specified model has a functional (i.e., behavioral simulation model) or physical (i.e., layout) description, it is an error if the specified ports are not directly connected to each other in the functional or physical model description. If the specified ports are not defined in the corresponding model description, the attributes are ignored.

Tools shall be able to traverse through the connected ports when performing driver and load analysis in the scope where the model is instantiated.

Example

Assume a macro cell with the following internal structure (see Figure G.1), where the cell has:

- two set of supplies: `vddA / vssA` and `vddB / vssB`
- `I1` drives logic powered by `vddA / vssA`
- `I2` has direct connection to port `O1` and `O2`
- `I3` drives logic powered by `vddB / vssB` and connection to port `O3`
- `I4` does not drive any logic internally
- `O4` is driven by logic powered by `vddB / vssB`

The following commands described the internal connection for input ports `I2` and `I3`, and output ports `O1`, `O2`, and `O3` of the cell:

```
set_port_attributes -ports {I2 O1 O2} -model cellX -feedthrough
set_port_attributes -ports {I3 O3} -model cellX -feedthrough
```

NOTE—Since input port `I3` also drives internal logic, it is allowed to have a **-receiver_supply** attribute set on port `I3` as well when a UPF power model is created for this cell.

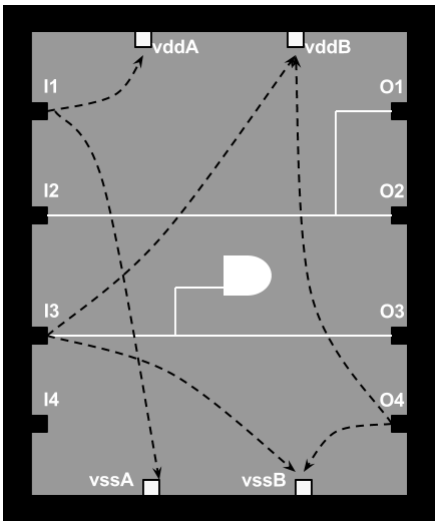

Figure G.1—Hard IP macro cell

In the following example,

```
set_port_attributes -ports {I2 O1} -model cellX -feedthrough
set_port_attributes -ports {I2 O2} -model cellX -feedthrough
```

the first command connects `I2` to `O1`, the second command connects `O2` to `I2`. As a result, `I2`, `O1`, and `O2` are all connected together, which is equivalent to the following:

```
set_port_attributes -ports {I2 O1 O2} -model cellX -feedthrough
```

Another way to specify the attribute is to use the corresponding UPF port attribute **UPF_feedthrough** directly, see 5.6.

G.2 Attributing unconnected ports of hard IP

In this case, the **set_port_attributes** command, with the **-unconnected** option (see 6.46), specifies a list of ports of a model that are not connected to any internal logic. If such a port is an input port, it means there is no logic within the model driven by the port; if such a port is an output port, it means there is no logic within the model driving the port. These ports shall not be associated with any other port attributes. This attribute also overwrites any default supply net or supply set association with respect to the specified ports, i.e., the specified ports are not associated with any supply net or supply set in UPF.

If the specified model has a functional (i.e., behavioral simulation model) or physical (i.e., layout) description, it is an error if the specified ports are connected to any logic or are part of the function definition of the functional or physical model description. If the specified ports are not defined in the corresponding model description, the attributes are ignored.

For simulation semantics, tools shall consider the signal driven by the specified ports as corrupted.

Another way to specify the attribute is to use the corresponding UPF port attribute **UPF_unconnected** directly, see 5.6.

Example

In the example from G.1, the input port `I4` is not connected to any internal logic. The following commands can be used to attribute that port as an unconnected one:

```
set_port_attributes -ports {I4} -model cellX -unconnected
```

Annex I

(informative)

Power-management cell modeling examples

This annex show how to model the power-management cells defined in Clause 7.

I.1 Modeling always-on cells

This subclause shows examples for how to model various types of always-on cells.

I.1.1 Types of always-on cells

An *always-on cell* is simply a library cell with more than one set of power and ground pins that can remain functional even when the supply to the rail-connected power or ground pin is switched off, as long as the non-switchable power or ground remains on. An always-on cell shall have at least a non-switchable power or a non-switchable ground pin defined.

A cell called always-on does not mean the cell can never be powered off. When the supply to the non-switchable power or ground of such cell is switched off, the cell becomes non-functional. In other words, the term *always-on* actually means relative always-on.

Any logic function can be implemented in the form of an always-on cell, such as an always-on buffer, always-on inverter, always-on `AND` gate, or even always-on flop. In the following subclauses, several different types of always-on cells are used as examples to describe how to use the **define_always_on_cell** command (see 7.2).

- Modeling a power-switched always-on buffer
- Modeling a ground-switched always-on buffer
- Modeling a power- and ground-switched always-on buffer
- Modeling a power-switched always-on flop with internal isolation

I.1.2 Modeling a power-switched always-on buffer

To model a power-switched always-on buffer, use the **define_always_on_cell** command (see 7.2) with the following options:

define_always_on_cell
    -cells *cells*
    -power *pin* **-power_switchable** *pin* **-ground** *pin*

In Figure I.1, a type of power-switched always-on buffer is shown. The cell's rail connection `VSW` is not used by the cell. The actual power of the cell comes from `VDD`, which needs to be routed separately. The following command models this type of cells:

```
define_always_on_cell
   -cells LP_Buf_Pow
   -power VDD -power_switchable VSW -ground VSS
```

The same command can also be used to describe any other type of power-switched always-on cells, such as an inverter, `AND` gate, etc.

LP_Buf_Pow

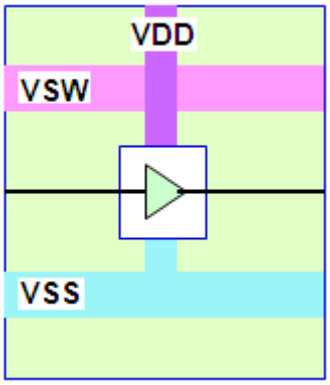

Figure I.1—Power-switched always-on buffer

I.1.3 Modeling a ground-switched always-on buffer

To model a ground-switched always-on buffer, use the **define_always_on_cell** command (see 7.2) with the following options:

define_always_on_cell
-cells *cells*
-power *pin* **-ground_switchable** *pin* **-ground** *pin*

In Figure I.2, a type of ground-switched always-on buffer is shown. The cell's rail connection `GSW` is not used by the cell. The actual ground of the cell comes from `VSS`, which needs to be routed separately. The following command models this type of cells:

```
define_always_on_cell
   -cells LP_Buf_Gnd
   -ground VSS -power VDD -ground_switchable GSW
```

The same command can also be used to describe any other type of ground-switched always-on cells, such as an inverter, `AND` gate, etc.

LP_Buf_Gnd

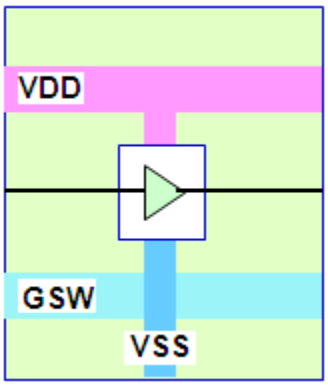

Figure I.2—Ground-switched always-on buffer

I.1.4 Modeling a power- and ground-switched always-on buffer

To model a power- and ground-switched always-on buffer, use the **define_always_on_cell** command (see 7.2) with the following options:

define_always_on_cell
    -cells *cells*
    -power_switchable *pin* **-ground_switchable** *pin*
    -power *pin* **-ground** *pin*

In Figure I.3, a type of power- and ground-switched always-on buffer is shown. The cell has both power and ground rail connections, `VSW` and `GSW`, respectively, but they are not used by the cell. The actual power and ground pins the cell come from `VDD` and `VSS`, which need to be routed separately. The following command models this type of cells:

```
define_always_on_cell
    -cells LP_Buf_Pow_Gnd
    -power VDD -ground VSS
    -power_switchable VSW -ground_switchable GSW
```

The same command can also be used to describe any other type of power- and ground-switched always-on cells such as an inverter, `AND` gate, etc.

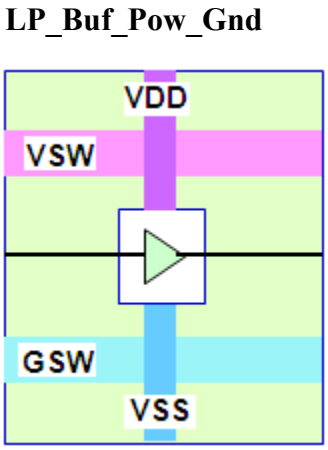

Figure I.3—Power- and ground-switched always-on buffer

I.1.5 Modeling a power-switched always-on flop with internal isolation

To model a power-switched always-on cell with internal isolation at some input pins, use the **define_always_on_cell** command (see 7.2) with the following options:

define_always_on_cell
    -cells *cells*
    -power *pin* **-power_switchable** *pin* **-ground** *pin*
    -isolated_pins *list_of_pin_lists* [**-enable** *expression_list*]

The always-on flip-flop cell in Figure I.4 has internal isolation at input pins `SE` and `SI` with the other input pin `ISO` as the control. The following command models this type of cells:

```
define_always_on_cell
    -cells LP_ff
```

```
-power VDD -power_switchable VSW -ground VSS \
-ioslated_pins { {SE SI} } -enable {!Iso}
```

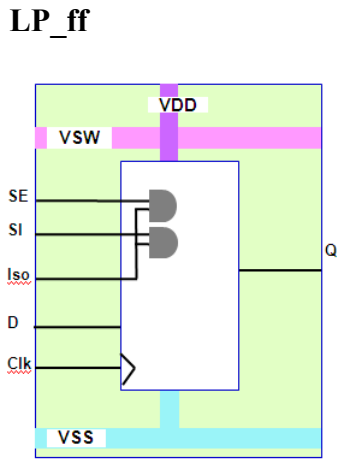

Figure I.4—Power-switched always-on flop with input isolation on pins SE and SI

I.2 Modeling cells with internal diodes

Cells with input pins connected to diodes need to be properly modeled to avoid electrical failure in a design with power-management. To model such cells, use the **define_diode_clamp** command (see 7.3) with the following options:

define_diode_clamp
-cells *cell_list*
-data_pins *pin_list*
[**-type** <**power** | **ground** | **both**>]
[**-power** *pin*] [**-ground** *pin*]

To describe the different type of diode connected pins shown in Figure I.5, use the following commands:

```
define_diode_clamp -cells cellA -data_pins in1 -type power -power VDD1
define_diode_clamp -cells cellB -data_pins in1 -type ground -ground VSS2
define_diode_clamp -cells cellC -data_pins in1 -type both \
    -power VDD1 -ground VSS2
define_diode_clamp -cells cellD -data_pins in1 -type power -power VDD
```

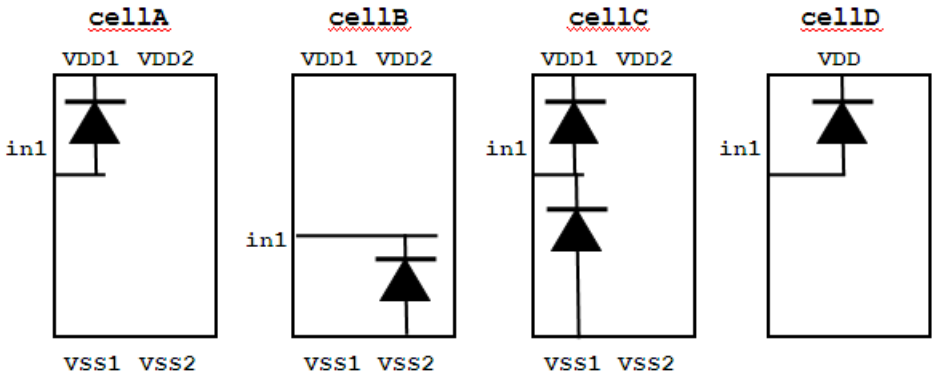

Figure I.5—Cells with different type of internal diodes

I.3 Modeling isolation cells

This subclause shows examples for how to model various types of isolation cells.

I.3.1 Types of isolation cells

Isolation logic is required when the leaf-drivers and leaf-loads of a net are in power domains that are not on and off at the same time, or because it is part of the design intent. The following is a list of the most typical isolation cells:

- Isolation cell to be placed in the unswitched domain
- Isolation cell to be used in a ground-switchable domain
- Isolation cell to be used in a power-switchable domain
- Isolation cells to be used in a a power- or ground-switchable domain
- Isolation cells without follow pins that can be placed in any domain
- Isolation cells without always-on power pins that can be placed in a switchable power domain
- Isolation cells without an enable pin
- Isolation clamp cell
- Isolation level-shifter combo cell

All types of isolation cells are defined using the **define_isolation_cell** command (see 7.4). The following subclauses indicate which command options to use for each type.

I.3.2 Modeling an isolation cell to be placed in the unswitched domain

To model an isolation cell to be placed in an unswitched domain, use the **define_isolation_cell** command (see 7.4) with the following options:

define_isolation_cell
-cells *cell_list*
-power *power_pin* **-ground** *ground_pin*
-valid_location on
{**-enable** *pin* | **-no_enable** <**high** | **low** | **hold**>}

Figure I.6 shows an `AND` cell that can be used for isolation purposes.