

**Berechnung von Verzögerung und Leistungsaufnahme
beim Entwurf von Chips**

Teil 1: System zur Berechnung von Verzögerung
und Leistungsaufnahme integrierter Schaltkreise (IC)
(IEC 61523-1:2001) Deutsche Fassung EN 61523-1:2002, Text in Englisch

DIN**EN 61523-1**

ICS 31.200

Delay and power calculation standards
Part 1: Integrated circuit delay and
power calculation systems
(IEC 61523-1:2001); German version EN 61523-1:2002

Normes de calculs de puissance et de temps de retard
Partie 1: Systèmes de calcul de puissance
et de temps de retard des circuits intégrés
(CEI 61523-1:2001); Version allemande EN 61523-1:2002

Die Europäische Norm EN 61523-1:2002 hat den Status einer Deutschen Norm.

Beginn der Gültigkeit

Die EN 61523-1 wurde am 2001-12-04 angenommen.

Nationales Vorwort

Für die vorliegende Norm ist das nationale Arbeitsgremium K 113 „Produktdatenmodelle, Informationsstrukturen, Dokumentation und graphische Symbole“ der DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE zuständig.

Die enthaltene IEC-Publikation wurde vom TC 93 „Design automation“ erarbeitet.

Das IEC-Komitee hat entschieden, dass der Inhalt dieser Publikation bis zum Jahr 2006 unverändert bleiben soll. Zu diesem Zeitpunkt wird entsprechend der Entscheidung des Komitees die Publikation

- bestätigt,
- zurückgezogen,
- durch eine Folgeausgabe ersetzt oder
- geändert.

Fortsetzung Seite 2
und 422 Seiten EN

DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE

Nationaler Anhang NA (informativ)

Zusammenhang mit Europäischen und Internationalen Normen

Für den Fall einer undatierten Verweisung im normativen Text (Verweisung auf eine Norm ohne Angabe des Ausgabedatums und ohne Hinweis auf eine Abschnittsnummer, eine Tabelle, ein Bild usw.) bezieht sich die Verweisung auf die jeweils neueste gültige Ausgabe der in Bezug genommenen Norm.

Für den Fall einer datierten Verweisung im normativen Text bezieht sich die Verweisung immer auf die in Bezug genommene Ausgabe der Norm.

Der Zusammenhang der zitierten Normen mit den entsprechenden Deutschen Normen ergibt sich, soweit ein Zusammenhang besteht, grundsätzlich über die Nummer der entsprechenden IEC-Publikation. Beispiel: IEC 60068 ist als EN 60068 als Europäische Norm durch CENELEC übernommen und als DIN EN 60068 ins Deutsche Normenwerk aufgenommen.

IEC hat 1997 die Benummerung der IEC-Publikationen geändert. Zu den bisher verwendeten Normnummern wird jeweils 60000 addiert. So ist zum Beispiel aus IEC 68 nun IEC 60068 geworden.

Berechnung von Verzögerung und Leistungsaufnahme beim Entwurf von Chips

Teil 1: System zur Berechnung von Verzögerung
und Leistungsaufnahme integrierter Schaltkreise (IC)
(IEC 61523-1:2001)

Delay and power calculation standards
Part 1: Integrated circuit delay and
power calculation systems
(IEC 61523-1:2001)

Normes de calculs de puissance
et de temps de retard
Partie 1: Systèmes de calcul de
puissance et de temps de retard
des circuits intégrés
(CEI 61523-1:2001)

Diese Europäische Norm wurde von CENELEC am 2001-12-04 angenommen. Die CENELEC-Mitglieder sind gehalten, die CEN/CENELEC-Geschäftsordnung zu erfüllen, in der die Bedingungen festgelegt sind, unter denen dieser Europäischen Norm ohne jede Änderung der Status einer nationalen Norm zu geben ist.

Auf dem letzten Stand befindliche Listen dieser nationalen Normen mit ihren bibliographischen Angaben sind beim Zentralsekretariat oder bei jedem CENELEC-Mitglied auf Anfrage erhältlich.

Diese Europäische Norm besteht in drei offiziellen Fassungen (Deutsch, Englisch, Französisch). Eine Fassung in einer anderen Sprache, die von einem CENELEC-Mitglied in eigener Verantwortung durch Übersetzung in seine Landessprache gemacht und dem Zentralsekretariat mitgeteilt worden ist, hat den gleichen Status wie die offiziellen Fassungen.

CENELEC-Mitglieder sind die nationalen elektrotechnischen Komitees von Belgien, Dänemark, Deutschland, Finnland, Frankreich, Griechenland, Irland, Island, Italien, Luxemburg, Malta, Niederlande, Norwegen, Österreich, Portugal, Schweden, Schweiz, Spanien, Tschechische Republik und dem Vereinigten Königreich.

CENELEC

Europäisches Komitee für Elektrotechnische Normung
European Committee for Electrotechnical Standardization
Comité Européen de Normalisation Electrotechnique

Zentralsekretariat: rue de Stassart 35, B-1050 Brüssel

Vorwort

Der Text des Schriftstücks 93/143/FDIS, zukünftige 1. Ausgabe von IEC 61523-1, ausgearbeitet von dem IEC TC 93 „Design automation“, wurde der IEC-CENELEC Parallelen Abstimmung unterworfen und von CENELEC am 2001-12-04 als EN angenommen.

Nachstehende Daten wurden festgelegt:

- spätestes Datum, zu dem die EN auf nationaler Ebene durch Veröffentlichung einer identischen nationalen Norm oder durch Anerkennung übernommen werden muss (dop): 2002-09-01
- spätestes Datum, zu dem nationale Normen, die der EN entgegenstehen, zurückgezogen werden müssen (dow): 2004-12-01

Diese Norm basiert auf IEEE Std P1481:1999; IEEE Standard for delay and power calculation systems.

Anerkennungsnotiz

Der Text der Internationalen Norm IEC 61523-1:2001 wurde von CENELEC ohne irgendeine Abänderung als Europäische Norm angenommen.

Contents

SECTION	PAGE
1. Overview	12
1.1 Scope	12
1.2 Purpose	12
1.3 Contents of this standard	13
2. References	14
3. Definitions	15
4. Acronyms and abbreviations	23
5. Delay and power calculation system architecture	24
5.1 Overview	24
5.2 Procedural interface	25
5.2.1 Global policies and conventions	25
5.2.2 Flow of control	26
5.3 DPCM - application relationships	26
5.3.1 Technology library	26
5.3.2 Subrule	27
5.4 Inter-operability	27
6. Delay Calculation Language (DCL)	28
6.1 Character set	29
6.2 Lexical elements	29
6.2.1 Whitespace	29
6.2.2 Comments	29
6.2.3 Tokens	29
6.2.4 Header names	39
6.2.5 Preprocessing directives	39
6.3 Name spaces of identifiers	40
6.4 Storage durations of objects	40
6.5 Scope of identifiers	40
6.6 Linkages of identifiers	41
6.6.1 EXPORT	41
6.6.2 IMPORT	41
6.6.3 FORWARD	41
6.6.4 Chaining of EXPOSE identifiers	41
6.7 DCL data types	41
6.7.1 Native data types	41
6.7.2 Array types	42
6.7.3 Derived data types	43
6.8 Type conversions	44
6.8.1 Implicit conversions	45
6.8.2 Explicit conversions	45
6.9 Operators	45
6.9.1 String prefix operator	46
6.9.2 Assignment operator	46
6.9.3 New operator	46

6.9.4	SCOPE operator	46
6.9.5	Purity operator	47
6.9.6	Timing propagation	47
6.9.7	Timing checks	47
6.9.8	Test mode operators	48
6.10	Expressions	50
6.10.1	Array subscripting	50
6.10.2	Statement calls	51
6.10.3	Assign variable reference	52
6.10.4	Store variable reference	52
6.10.5	Mathematical expressions	52
6.10.6	Logical expressions and operators	54
6.10.7	Pin range	55
6.10.8	Embedded C code expressions	57
6.11	Computation order	58
6.11.1	Mathematical expressions	58
6.11.2	Logical expressions	58
6.11.3	Passed parameters	59
6.11.4	WHEN clause	59
6.11.5	REPEAT - UNTIL clause	59
6.12	DCL statements	59
6.12.1	Clauses	59
6.12.2	Modifiers	62
6.12.3	Prototypes	63
6.12.4	Statement failure	66
6.12.5	Interfacing statements	67
6.12.6	Calculation statements	68
6.12.7	METHOD statement	72
6.13	Tables	73
6.13.1	TABLEDEF statement	73
6.13.2	Table visibility rules	75
6.13.3	TABLE statement	76
6.13.4	Static tables	76
6.13.5	Dynamic tables	78
6.13.6	Dynamic table manipulation	79
6.13.7	Lookup table	83
6.14	Library control statements	86
6.14.1	Meta-variables	86
6.14.2	SUBRULE statement	86
6.14.3	SUBRULES statement	88
6.14.4	TECH_FAMILY statement	90
6.15	Modeling	91
6.15.1	Model organization	91
6.15.2	MODELPROC statement	92
6.15.3	SUBMODEL statement	94
6.15.4	Modeling statements	94
6.16	Embedded C code	111
6.17	Definition of a subrule	111
7.	Power modeling and calculation	113
7.1	Power overview	114
7.2	Caching state information	115
7.2.1	Initializing the state cache	115

7.2.2	State cache lifetime	115
7.3	Caching load and slew information	115
7.3.1	Loading the load and slew cache	116
7.3.2	Load and slew cache lifetime	116
7.4	Simultaneous switching events	116
7.5	Partial swing events	117
7.6	Power calculation	117
7.7	Accumulation of power consumption by the design	119
7.8	Group pin list syntax and semantics	119
7.8.1	Syntax	119
7.8.2	Semantics	120
7.8.3	Example	120
7.9	Group condition list syntax and semantics	120
7.9.1	Syntax	121
7.9.2	Semantics	121
7.9.3	Example	121
7.10	Sensitivity list syntax and semantics	122
7.10.1	Syntax	122
7.10.2	Semantics	122
7.10.3	Example	122
7.11	Group condition language	123
7.11.1	Syntax	123
7.11.2	Semantics	123
7.11.3	Condition expression operator precedence	125
7.11.4	Condition expressions referencing pin states and transitions	126
7.11.5	Semantics of nonexistent pins	126
8.	Procedural Interface (PI)	128
8.1	Overview	128
8.1.1	DPCM	128
8.1.2	Application	128
8.1.3	libdcmlr	128
8.2	Control and data flow	129
8.3	Architectural requirements	129
8.4	Data ownership technique	129
8.4.1	Persistence of data passed across the PI	129
8.4.2	Data cache guidelines for the DPCM	130
8.5	Application/DPCM interaction	130
8.5.1	Application initializes message/memory handling	130
8.5.2	Application loads and initializes the DPCM	130
8.5.3	Application requests timing models for cell instances	131
8.5.4	Model domain issues	131
8.5.5	DPCM invokes application modeling callback functions	131
8.5.6	Application requests propagation delay	132
8.5.7	DPCM calls application EXTERNAL functions	133
8.6	Re-entry requirements	133
8.7	Application responsibilities when using a DPCM	133
8.7.1	Standard structure rules	133
8.7.2	User object registration	133
8.7.3	Selection of early and late slew values	133
8.8	Application use of the DPCM	135
8.8.1	Initialization of the DPCM	135
8.8.2	Use of the DPCM	136

8.8.3	Termination of DPCM	137
8.9	DPCM library organization	137
8.9.1	Multiple technologies	137
8.9.2	Model names	138
8.10	DPCM error handling	138
8.11	C level language for EXPOSE and EXTERNAL functions	138
8.11.1	Integer return code	138
8.11.2	The Standard Structure pointer	139
8.11.3	Result structure pointer	139
8.11.4	Passed arguments	139
8.11.5	DCL array indexing	140
8.11.6	Conversion to C data types	140
8.11.7	include files	140
8.12	PIN and BLOCK data structure requirements	141
8.13	DCM_STD_STRUCT Standard Structure	142
8.13.1	Alternate semantics for Standard Structure fields	145
8.13.2	Reserved fields	145
8.13.3	Standard Structure value restriction	145
8.14	DCMTransmittedInfo structure	146
8.15	Environment or user variables	146
8.16	PI functions summary	146
8.16.1	Expose functions	146
8.16.2	External functions	149
8.16.3	Implicit functions	151
8.16.4	PI function table description	154
8.17	PI function descriptions	156
8.17.1	Interconnect loading related functions	156
8.17.2	Interconnect delay related functions	162
8.17.3	Functions accessing netlist information	165
8.17.4	Functions exporting limit information	186
8.17.5	Functions getting/setting model information	188
8.17.6	Functions importing instance name information	199
8.17.7	Process information functions	202
8.17.8	Miscellaneous standard interface functions	203
8.17.9	Power related functions	213
8.17.10	Array manipulation functions	225
8.17.11	Initialization functions	229
8.17.12	Calculation functions	240
8.17.13	Modeling functions	245
8.18	86Standard structure (dcmstd_stru.h) file	257
8.19	Standard macros (dcmstd_macros.h) file	273
8.20	Standard interface structures (dcmintf.h) file	276
8.21	Standard loading (dcmload.h) file	280
8.22	Standard debug (dcmdebug.h) file	282
8.23	Standard array (dcmgarray.h) file	297
8.24	DCM user array defines (dcmuarray.h) file	301
8.25	Standard platform-dependency (dcmpltfm.h) file	303
8.26	Standard state variables (dcmstate.h) file	310
8.27	Standard table descriptor(dcmutab.h)	313
9.	Parasitics	313
9.1	Introduction	314
9.2	Targeted applications for SPEF	314

9.3	SPEF specification	314
9.3.1	Grammar	314
9.3.2	File syntax	316
9.3.3	Escaping rules	323
9.3.4	Comments	324
9.3.5	File semantics	324
9.4	Examples	340
9.4.1	Basic D_NET file	340
9.4.2	Basic R_NET file	343
9.4.3	R_NET with poles and residues plus name mapping	344
9.4.4	D_NET with triplet par_value	346
9.4.5	R_NET with poles and residues plus triplet par_value	349
9.4.6	Merging SPEF files	350
10.	Physical design exchange	356
10.1	Introduction	357
10.1.1	Scope	357
10.1.2	Targeted applications	357
10.2	PDEF specification	358
10.2.1	PDEF grammar	358
10.2.2	PDEF file syntax	360
10.2.3	Comments within a PDEF file	364
10.2.4	PDEF name semantics	364
10.2.5	PDEF file semantics	367
10.2.6	Attributes	378
10.3	Examples	398
10.3.1	Escaping	399
10.3.2	Clusters	399
10.3.3	Global routing	403
10.3.4	Symbolic placement constraints using PDEF	404
A.	Implementation requirements	407
B.	Calculation of total load capacitance in the DPCS	409
C.	Hold Control	412
D.	Bibliography	417

List of Figures

FIGURE	PAGE
5-1 High-level DPCS architecture	24
5-2 High-level DPCS architecture linkage structure	26
8-1 DPCM/application procedural interface	129
8-2 PIN and PINLIST	142
8-3 PI function table example	155
8-4 Parallel drivers example	168
8-5 Capacitance value example	173
8-6 Passed and receiver pin pointers example	184
8-7 Integer LSB example	215
8-8 Bias calculation	244
8-9 Clock separation	244
8-10 Different edges	245
8-11 Sample MODELPROC results	253
8-12 Additional MODELPROC results	254
9-1 SPEF targeted applications	314
10-1 PDEF targeted applications	358
10-2 Hierarchical routes	375
10-3 Clusters cross logic hierarchy	400
10-4 Illustration of the chip used in the example	400
10-5 A global route	404
10-6 Symbolic cell and cluster placement	405
10-7 Symbolic pin and net placement	406
C.1 Transparent latch pair feedback	412
C.2 Overlap period	413
C.3 Padding the paths	414
C.4 Stable overlap period	414
C.5 Hold Control modeling	415
C.6 Snipped feedback line	416

List of Tables

TABLE		PAGE
6-1	Keywords	31
6-2	DCL predefined references to Standard Structure fields	32
6-3	DCL compiler generated predefined identifiers	36
6-4	Edge types and conversions	37
6-5	Propagation mode conversions	37
6-6	Calculation mode conversions	38
6-7	TEST_TYPE conversions	38
6-8	VAR array modifier	43
6-9	Purity operator	47
6-10	Timing resolution modes	47
6-11	Test mode operators	48
6-12	Mathematical operators	53
6-13	Logical operators	55
6-14	Mathematical operator precedence (high to low)	58
6-15	Logical operator precedence (high to low)	58
6-16	Validity of predefined identifiers for STORE clause	106
7-1	PinName_Identifier semantics	125
7-2	PinName_Level semantics	125
7-3	PinName_State semantics	125
7-4	Condition expression operators	126
8-1	Interaction between multiple technologies and application	138
8-2	Return code most significant byte	139
8-3	Return code least significant bytes	139
8-4	Data types defined in DCL and C	140
8-5	Header files	141
8-6	Predefined macro names	142
8-7	Alternate semantics for Standard Structure fields	145
8-8	EXPOSE functions	146
8-9	EXTERNAL functions	149
8-10	libdcmlr functions	151
8-11	Run-time library	152
8-12	Calculation functions	154
8-13	Modeling functions	154
8-14	Standard Structure field semantics	155
8-15	Mask encoding	216
8-16	Mode propagation operators	247
8-17	Mode computation operators for delay and slew	247
8-18	Mode operator enumerators for check	248
8-19	Enumeration pairs	248
8-20	Edge propagation communication with DPCM	250
9-1	Design flow values	325
9-2	conn_attr types	329
10-1	Netlist type identifiers	368
10-2	Design flow identifiers	369
10-3	Gate type attributes	380
10-4	Gate restriction attributes	381

10-5	Pin and gate_pin attributes	383
10-6	Node type attributes.	384
10-7	Net type attributes	386
10-8	Net restriction attributes	387
10-9	Route restriction attributes	387
10-10	Route connectivity attributes.	388
10-11	Cluster restriction attributes	390
10-12	Cluster connectivity attributes.	393
10-13	Cell type attributes.	394
10-14	Cell restriction attributes	395
10-15	Spare cell restriction attributes	396
10-16	Pbus and nbus type attributes	397
10-17	Pnet type attributes	397
10-18	Pnet restriction attribute pre-defined values	398
10-19	Gap and skip type attributes	398

Introduction

The objective of the Delay and Power Calculation System (DPCS) is to make it possible for integrated circuit designers to *consistently* calculate chip delay and power across Electronic Design Automation (EDA) applications and for integrated circuit vendors to express delay and power information only *once* per technology while enabling sufficient EDA application accuracy.

This is accomplished by a coordinated set of standards which support a standard method to describe timing and power characteristics of integrated circuit design units (cells and higher level design elements); a standard method for EDA applications to calculate chip design instance specific delay, slew, and power for logic and interconnects; and standard file formats to exchange chip parasitic and cluster information.

A major integrated circuit design problem is to ensure the chip meets the designer's timing and power requirements. At this time in the EDA industry, there are numerous methodologies — many incompatible or inconsistent with each other — for representing timing and power information. The DPCS responds to the urgency to improve chip design methods.

The DPCS includes all aspects and uses of delay and power calculation (including synthesis, floorplanning, and simulation) during both pre-layout and post-layout phases of chip design. A fundamental requirement placed on the DPCS is that it work equally well with Verilog HDL and VHDL. In this draft of the standard, the scope of the DPCS is limited to integrated circuit designs.

The DPCS gains its leverage out of an effort by Silicon Integration Initiative (Si2) to standardize on contributed technology from IBM in the form of a Delay Calculation Language (DCL) and its companion procedural interface (PI), along with an effort by Open Verilog International (OVI) to standardize on file formats to exchange chip parasitic and cluster information in the form of a Physical Design Exchange Format (PDEF) and a Standard Parasitic Exchange Format (SPEF). SPEF is based on the Standard Parasitic Format (SPF) technology supplied by Cadence Design Systems. PDEF is based on technology provided by Synopsys.

Since each of these specifications is based on existing technology which has proven itself in industry, the time to introduce this standard into industry is greatly reduced over what would have been required for a newly created standard. The industry owes a great deal of gratitude to these companies for contributing these technologies.

The P1481 Working Group is composed of four subgroups: Architecture, Language, Parasitics and Clustering, and Power. The members of each of these groups are members of the P1481 Working Group. The P1481 Working Group has a vice-chair liaison with Japan to coordinate development and review of the specification there. The IEC TC93 WG 2 also participates in the review and development of this specification.

In addition to numerous individual contributors, both OVI, VHDL International (VI), and Si2 have played significant roles in the development of this standard and funding of the technical documentation.

DELAY AND POWER CALCULATION STANDARDS –

Part 1: Integrated circuit delay and power calculation systems

1. Overview

The Delay and Power Calculation System (DPCS) standard is a coordinated set of standards which support a standard method to describe timing and power characteristics of integrated circuit design units (cells and higher level design elements); a standard method for EDA applications to calculate chip design instance specific delay, slew, and power for logic and interconnects; and standard file formats to exchange chip parasitic and cluster information. The four distinct standard specifications covered in this document include:

- A description language for timing and power modeling (DCL).
- A software procedural interface (PI) for communications between EDA applications and compiled libraries of DCL descriptions.
- A standard file exchange format for parasitic information about the chip design (SPEF).
- A standard file exchange format for floorplan cluster information relative to the chip design (PDEF).

1.1 Scope

As stated in the introduction, the scope of the DPCS standard is to make it possible for integrated circuit designers to analyze chip timing and power consistently across a broad set of EDA applications, for integrated circuit vendors to express timing and power information once (for a given technology), and for EDA vendors to meet their application performance and capacity needs. The intended use for these standards is integrated circuit timing and power. The standard may be applied to both unit logic cells supplied by the integrated circuit vendor and logical macros defined by the integrated circuit designer. Although this specification is written towards the integrated circuit supplier and EDA developer, its application applies equally well to representation of timing and power for designer defined macros (or hierarchical design elements).

These specifications make it *possible* to achieve consistent timing and power results, but do not *guarantee* it. They provide for a single executable software program which computes delay and power based on integrated circuit vendor-supplied algorithms (or designer-supplied algorithms for macros), but does not guarantee EDA applications correctly communicate the design-specific information required for these algorithms. By specifying standard exchange formats for parasitic data and floorplanning information, the standard provides a marked improvement over design environments with no such standards. However, it is the responsibility of the EDA application to correctly correlate the information between these standard exchange files and the actual design. These specifications also do not detail how the information contained within the standard exchange files shall be obtained.

1.2 Purpose

As feature sizes for chips shrink below 0.5µm, interconnect delay effects outweigh those of the logic cells. This means placement of cells and wire routing of the interconnects become as important a factor as the type of cell drivers and receivers on the interconnect. As a result, EDA logic design applications (such as synthesis) now need to interact closely with physical design applications (such as floorplanning and layout). Applications that before could consider only simple delay and power models now need to deal with complex delay and power equations. The designer now needs EDA applications which can be directed to specific timing constraints, and provide consistent and accurate characterization of a chip's timing and power before it is manufactured. Plus, due to the complexities of the delay and power equations, the integrated circuit vendor needs to have control of application calculations and not be restricted by unique characteristics of the broad set of applications demanded by the customers (the designers).

Over the past few years it has become increasingly apparent modern VLSI design is no longer bounded only by timing and area constraints. Power has become significantly more important. In an era of hand-held devices, ranging from mobile computing to wireless communication systems, managing and controlling power takes on an important role.

Several benefits can be attained from low power designs in addition to extended battery life. Low power devices often run at lower junction temperature which leads to higher reliability and lower cost cooling systems. There are also several challenges for calculation and modeling of power (and delay) in deep-submicron (less than 0.25µm) designs. Now, EDA tools can accurately calculate and model power by using this DPCS standard.

1.3 Contents of this standard

The organization of the remainder of this standard is:

- Clause 2 (References) provides references to other applicable standards that are assumed or required for DPCS.
- Clause 3 (Definitions) defines terms used throughout the different specifications contained in this standard.
- Clause 4 (Acronyms and abbreviations) defines the acronyms used in this standard.
- Clause 5-1 (High-level DPCS architecture) gives an overview of how all of the elements of this standard fit together to achieve the DPCS goals.
- Clause 6 (Delay Calculation Language (DCL)) provides the specification for the language used to model timing and power.
- Clause 7 (Power modeling and calculation) provides the specification for modeling power and the interaction between an application and a DPCM for power calculations.
- Clause 8 (Procedural Interface (PI)) provides the specification of the functions used to communicate between EDA applications and the DCL based libraries as well as their software interface.
- Clause 9 (Parasitics) provides the specification for the exchange of parasitics which are required to calculate accurate delays.
- Clause 10 (Physical design exchange) provides the specification for the exchange of floorplan clustering information which is used to provide more accurate timing calculations based on approximate layout.
- Annexes. Following Clause 10 are a series of normative and informative appendices.

2. References

This standard shall be used in conjunction with the following publications. When the following standards are superseded by an approved revision, the revision shall apply.

ISO/IEC 9899:1990, Programming Languages — C

3. Definitions

For the purposes of this recommended practice, the following terms and definitions apply. IEEE Std 100-1992, The New IEEE Standard Dictionary of Electrical and Electronics Terms, should be referenced for terms not defined in this standard.

3.1 application, EDA application: Any software program that interacts with the *delay and power calculation module (DPCM)* through the *procedural interface (PI)* to compute instance-specific timing values. Examples include batch delay calculators, synthesis tools, floor-planners, static timing analyzers, etc. *See also: delay calculation module, procedural interface.*

3.2 arc: *See: timing arc.*

3.3 argument: The value or the address of a data item passed to a function or procedure by the caller.

3.4 timing calculation, delay calculation: The process of calculating values for the delays and timing checks associated with the physical primitives (*cells*) of an integrated circuit design, or part of an integrated circuit design, and their interconnections.

3.5 back-annotation: The annotation of information from further downstream steps (towards fabrication) in the design process. *See also: back-annotation file.*

3.6 back-annotation file: A file containing information to be read by a tool for the purpose of back-annotation, for example Physical Design Exchange Format (PDEF) and Standard Parasitic Exchange Format (SPEF) files. *See also: back-annotation, timing annotation.*

3.7 bias: The time difference between the data arrival time and a specified signal edge (e.g., of a clock). Also, the BIAS clause used in a CHECK statement.

3.8 bidirectional: A *pin* or *port* which can place logic signals onto an *interconnect* and receive logic signals from it (i.e., act both as a driver and a receiver).

3.9 bus: In PDEF, a physical collection of *nets* and/or *pnets*, or of *pins* and/or *nodes*. If the items collected in the PDEF bus are logical, the PDEF bus may or may not correspond to a logical bus described in the netlist.

3.10 C-effective: A capacitance value, often computed as an approximation to an Resistor/Inductor/Capacitor (RLC) network or a π -model, that characterizes the admittance of an *interconnect* structure at a particular driver. The reduction of real parasitics and pin capacitances to a C-effective allows the calculation of delay and slew values from cell characterization data which assumes a pure capacitive output load. *Syn: effective capacitance.*

3.11 cell: A primitive in an integrated circuit library. For the purposes of this specification, “primitive” means the timing properties of the cell are directly described in the *DPCM* without reference back to the application for the internal structure of the cell. This primitiveness typically is a result of the characterization of that cell by the semiconductor vendor, but it may instead be a result of the construction of a timing model for a sub-circuit by the application and its loading into the *DPCM* at run-time. For PDEF, **cell** refers to a logical **instance** and a library primitive is called a **gate**.

The term cell can arise in the context of the abstraction of a type of cell available in the library or in the concrete selection and placement of a cell in the final design. If the context is not clear, the terms *cell type* and *cell instance* (or just *instance*) shall be used. *See also: cell type, instance.*

3.12 cell type: Name used to identify a particular cell in the library.

- 1 3.13 **cluster**: A grouping of cell instances and/or clusters which are constrained to each other due to physical location or some other shared characteristic(s). It is not valid to have a cell instance explicitly made a member of more than one cluster. *Syn*: **partition**, **region**.
- 5 3.14 **column**: In a PDEF *datapath cluster*, a *cluster of cell, spare_cell*, and/or *cluster instances* placed or constrained to be placed in the vertical (Y-axis) direction. *See also*: **row**, **datapath**.
- 10 3.15 **constraint**: A timing property of a design which is supplied as a goal or objective to an EDA tool, such as logic synthesis, floorplanning, or layout. The tool shall not start out with a fixed design implementation; it shall build or modify the design to meet the constraint. *See also*: **timing check**.
- 3.16 **datapath**: A type of PDEF *cluster* which contains *rows* and/or *columns* of *cluster, cell, skip*, and/or *spare_cell instances*. A PDEF datapath typically corresponds to structured logic. *See also*: **row**, **column**.
- 15 3.17 **delay**: The time taken for a digital signal to propagate between two points;
- 3.18 **delay arc**: *See*: **timing arc**.
- 20 3.19 **delay calculation language (DCL)**: The programming language used to calculate instance-specific timing data. DCL contains high-level constructs which can refer to the aspects of the design topology that influence timing and also express the sequence of calculations necessary to compute the desired delay and timing check limit values.
- 25 3.20 **delay calculation language compiler**: A software program, used in conjunction with a C compiler, that reduces *DCL* from ASCII text to computer executable format. *See also*: **delay and power calculation module**.
- 30 3.21 **delay and power calculation module (DPCM)**: A *delay and power calculation system (DPCS)* -compliant software component supplied by a semiconductor vendor that is responsible for computing instance-specific timing data under control of an EDA application. The DPCM is loaded into memory at run-time and linked to the application via the PI. A DPCM typically is created from DCL subrules compiled by the DCL compiler and linked together with run-time support modules.
- 35 3.22 **delay and power calculation system (DPCS)**: The complete system detailed in this specification: the DCL language, the PI for delay and power calculations, and text formats for physical design and parasitic information.
- 3.23 **delay equation**: Any mathematical expression describing cell delay or interconnect delay.
- 40 3.24 **driver**: A pin of a cell instance that, in the current context, is placing or can place a signal onto an interconnect structure.
- 3.25 **early mode**: The very first edge that propagates through a given cone of logic.
- 45 3.26 **fanout**: The pin count of a net (the number of pins connected to the net), minus one. This definition includes all input, output, and bidirectional *pins* on the net with the sole exception of one pin (assumed to be related to the particular timing arc currently of interest). Although less fundamental than *pin count*, fanout is frequently used in the definition of wireload models.
- 50 3.27 **forward annotation**: The annotation of information from further upstream (earlier in the design flow) in the design process. *See also*: **forward annotation file**.
- 3.28 **forward annotation file**: A file containing information to be read by a tool for the purpose of forward annotation, for example an SDF file containing PATHCONSTRAINTS. *See also*: **forward annotation**.

- 1 3.29 **function, PI function**: One of the C functions that comprise the DPCS procedural interface.
- 3.30 **gap**: In PDEF, spacing between *rows* and/or *columns* in a *datapath*.
- 5 3.31 **gate**: In PDEF, the physical abstraction of an library primitive.
- 3.32 **hard macro**: A cluster whose cell placements relative to each other are fixed. Often the interconnect routing between the cells is also fixed and a parasitics file describing the *interconnect* is available for the hard macro. The location of the hard macro in the floorplan may or may not be fixed.
- 10 3.33 **hard region**: A cluster which has defined physical boundaries in a floorplan. All cells contained in the cluster shall be placed within the boundaries of the cluster.
- 15 3.34 **hierarchical instance**: The concrete appearance of a design unit at some hierarchical level. Because higher-level design units may be instantiated multiple times, a single such appearance may give rise to multiple instances of the lower-level design units within it. Where instances are referred to as “occurrences”, hierarchical instances are referred to simply as *instances*.
- 20 3.35 **hold timing check**: A timing check that establishes only the end of the stable interval for a *setup/hold timing check*. If no setup timing check is provided for the same arc, *transitions*, and state, the stable interval is assumed to begin at the reference signal transition and a negative value for the *hold time* is not meaningful. *See also*: **setup/hold timing check**.
- 25 3.36 **hold time**: *See*: **setup/hold timing check, nochange timing check**.
- 3.37 **software implementation-defined behavior**: Behavior, for a correct program construct and correct data, that depends on the software implementation and that each implementation shall document. The range of possible behaviors is delineated by the standard.
- 30 3.38 **software implementation limits**: Restrictions imposed by an implementation.
- 3.39 **input**: A *pin* or *port* which shall only receive logic signals from a connected *net* or *interconnect* structure.
- 35 3.40 **instance, cell instance**: A particular, concrete appearance of a cell in the fully expanded (flattened, unfolded, elaborated) design description of an integrated circuit, also referred to elsewhere as an “occurrence.” An instance is a “leaf” of the unfolded design hierarchy. In PDEF, this is a physical *cluster* or a logical *cell*. *See also*: **cell, cell type, cluster, hierarchical instance**.
- 40 3.41 **interconnect**: A collective term for structures (in an integrated circuit) which propagate a signal between the pins of cell instances with as little change as possible. These structures include metal and polysilicon segments, vias, fuses, anti-fuses, etc. But, interconnect shall not include such structures if they occur as part of the fixed layout of a cell.
- 45 3.42 **late mode**: The very last edge that propagates through a given cone of logic.
- 3.43 **layer**: In PDEF, a particular level of interconnect on which a logical or physical pin is located.
- 50 3.44 **library (integrated circuit)**: A collection of circuit functions, implemented in a particular integrated circuit technology, which an integrated circuit designer or EDA synthesis application can select in order to implement a design. *See also*: **cell**.
- 3.45 **library (software)**: A collection of object code units that may be linked, either statically or at run-time, with other libraries and/or object code to produce a software program.

- 1 3.46 **library control statements**: These statements control the logical organization and loading of subrules in a technology library. *See also*: **subrule**, **technology library**.
- 5 3.47 **load-dependent delay**: That part of a delay through a cell instance attributed to the admittance (load) presented to the arc sink pin and the internal impedance of the output.
- 10 3.48 **mesh table**: A multi-dimension table which defines every type of delay model in terms of discrete points. Each point represents a delay value in terms of several cell parameters or interconnect parameters. The delay calculation module is expected to interpolate between these points based on a mathematical expression defined by the technology file.
- 15 3.49 **(to) model a cell**: The creation of a specific elaboration of a model using `modelSearch`.
- 20 3.50 **modeling procedures**: Describe the action of a circuit with respect to timing and power. These actions include creating segments and nodes, determining the propagation properties, and setting the delay and slew equations to use.
- 25 3.51 **modeling statements**: DCL statements which map cell configurations to modeling procedures.
- 30 3.52 **net, net instance**: An abstraction expressing the idea of an electrical connection between various points in a design. In a hierarchical representation of the design, nets can occur at all levels and may connect to pins of lower hierarchical levels (including cell instances), *ports* of the current hierarchical level and each other. In a flattened (unfolded and elaborated) design, electrically connected nets are collapsed and each net instance corresponds to an unique interconnect structure in the implementation.
- 35 3.53 **nochange timing check**: A timing check similar to a *setup/hold timing check* except the setup and hold times are referred to opposite transitions of the reference signal. The stable interval is extended to include the period between these *transitions*, i.e., the time for which the reference signal stays in a specified state. This *timing check* is frequently applied to memory and latch-banks to establish the stability of the address or select inputs before, during, and after the write pulse.
- 40 3.54 **node**: A conceptual point (through which logic signals pass) which has been identified as an aid to modeling the timing properties of a cell but may not correspond to any physical structure. In PDEF, this is a physical pin which does not correspond to a logical structure.
- 45 3.55 **nugget**: A data structure used in the PI for rapid switching between technologies.
- 50 3.56 **output**: A *pin* or *port* shall only to place logic signals onto a connected net or interconnect structure.
- 3.57 **parameter**: A data item required for the calculation of some result.
- 3.58 **parasitics**: Electrical properties of a design (resistance, capacitance, and impedance) that arise due to the nature of the materials used to implement the design.
- 3.59 **period timing check**: A timing check which specifies the allowable time between successive periods of a signal.
- 3.60 **periphery**: The outer part of an integrated circuit where instances of cell types designed specifically to interface the internal circuitry to the “outside world” are placed. This part includes “pad” cells (which are input and output buffers) and power and ground pads; it may also include test circuitry, such as boundary scan cells.
- 3.61 **π -model**: A simplification of a general RLC network that represents the driving point admittance for an interconnect.

3.62 **pin**: A terminal point where an interconnect structure makes electrical contact with the fixed structures of a cell instance or the conceptual point where a net connects to a lower level in the design hierarchy.

3.63 **pin count**: The number of cell instance pins that an interconnect structure visits, including all input, output, and bidirectional pins. Pin count is the number of “places” the interconnect goes to on the chip.

3.64 **pnet**: A physical net which has no correspondence to the logical function of the design, such as a *route* segment which is reserved for future *routes* across a hard macro, or a power net not described in the design netlist.

3.65 **pole**: The complex frequency where a Laplace Transform is infinite. Combined with residues, this is a convenient mathematical notation for the impedance or transfer function of a passive circuit, such as an RLC circuit, since poles above this frequency can be ignored in calculations without significant loss of accuracy.

3.66 **port**: A conceptual point at which a cell or a hierarchical design unit makes its interface available to higher levels in the design hierarchy.

3.67 **primary input**: The point where a logic signal arrives at the boundary of the design as currently known to an EDA application. For a complete integrated circuit design, for example, this point is the metal pad of an input or bidirectional pad cell.

3.68 **primary output**: The point where a logic signal leaves the design as currently known to an EDA application. For a complete integrated circuit design, for example, this point is the metal pad of an output or bidirectional pad cell.

3.69 **procedural interface, PI**: The set of C functions used by an application and a DPCM to exchange information and determine the timing calculation for a design.

3.70 **pulse width timing check**: A timing check that specifies the minimum time a signal shall remain in a specified state once it has transitioned to that state.

3.71 **RC, RC time constant**: The product of some resistance and some capacitance (having the dimensions of time) or a time constant computed in some other way. *Syn*: **Elmore delay**.

3.72 **receiver**: A pin of a cell instance that is receiving or can receive a signal from an interconnect structure.

3.73 **recovery/removal timing check**: A timing check that establishes an interval with respect to a reference signal transition during which an asynchronous control signal may not change from the active to inactive state. This timing check is frequently applied to flip-flops and latches to establish a stable interval for the set and reset inputs with respect to the active edge of the clock or the active-to-inactive transition of the gate.

Two limit values are necessary to define the stable interval. The recovery time is the time before the reference signal transition when the stable interval begins. The removal time is the time after the reference signal transition when the stable interval ends.

If the asynchronous control signal goes inactive during the stable interval, it is unknown whether the flip-flop or latch takes on the state of the data input, remains set, or is reset.

3.74 **recovery time**: *See*: **recovery/removal timing check**.

3.75 **recovery timing check**: A timing check that establishes only the beginning of the stable interval for a recovery/removal timing check. If no *removal timing check* is provided for the same arc, transitions, and state, the stable interval is assumed to end at the reference signal transition and a negative value for the recovery time is not meaningful. *See also*: **recovery/removal timing check**.

3.76 **region**: A region pertains to a particular physical section or block of a floorplan. *See*: **cluster**.

3.77 **removal time**: *See*: **recovery/removal timing check**.

3.78 **removal timing check**: A timing check that establishes only the end of the stable interval for a recovery/removal timing check. If no recovery timing check is provided for the same arc, transitions, and state, the stable interval is assumed to begin at the reference signal transition and a negative value for the removal time is not meaningful. *See also*: **recovery/removal timing check**.

3.79 **residue**: The value of $\lim_{s \rightarrow s_0} (s - s_0) \times F(s)$, where $F(s)$ has the complex pole s_0 . *See also*: **pole**.

3.80 **return code**: A value returned by a function indicating whether the function completed successfully. If the function did not complete successfully, it may return a nonzero return code; the exact value may indicate one of several possible severity conditions: informational, warning, error, severe, terminal error, etc.

3.81 **route, global route**: In PDEF, the physical description of interconnect routing between logical and physical pins of *cell*, *spare_cell*, and/or *cluster instances*.

3.82 **row**: In a PDEF *datapath cluster*, a *cluster* of *cell*, *spare_cell*, and/or *cluster instances* placed or constrained to be placed in the vertical (Y-axis) direction. *See also*: **column, datapath**.

3.83 **scalar**: An integer constant.

3.84 **sequence point**: A certain point in the execution sequence of a program where all side effects of previous evaluations are complete and no side effects of subsequent evaluations have occurred. (Refer to ISO/IEC 9899:1990, Section 5.1.2.3, page 7.)

3.85 **segment**: A portion of an interconnect structure treated as a unit for the purposes of extracting or estimating its electrical properties. *See also*: **parasitics**.

3.86 **setup/hold timing check**: A timing check that establishes an interval with respect to a reference signal transition during which some other signal may not change value. This timing check is frequently applied to flip-flops and latches to establish a stable interval for the data input with respect to the active edge of the clock or the active-to-inactive transition of the gate.

Two limit values are necessary to define the stable interval. The setup time is the time before the reference signal transition when the stable interval begins and shall be negative if the stable interval begins after the reference signal transition. The hold time is the time after the reference signal transition when the stable interval ends and shall be negative if the stable interval ends before the reference signal transition.

If the data signal changes during the stable interval, the reliability of the resulting state of the flip-flop or latch is unknown.

3.87 **setup time**: *See*: **setup/hold timing check, nochange timing check**.

3.88 **setup timing check**: A timing check that establishes only the beginning of the stable interval for a setup/hold timing check. If no hold timing check is provided for the same arc, transitions, and state, the stable interval is assumed to end at the reference signal transition and a negative value for the setup time is not meaningful. *See also*: **setup/hold timing check**.

3.89 **shared port**: An output or bidirectional port where some other output port of the cell derives its logic function. The output load at a shared port affects not only the delay to that port itself, but also the delay to any ports sharing it.

- 1 3.90 **sink, sink pin**: The sink pin is the end of a delay arc, i.e. the destination of the logic signal. For arcs across cell instances, the sink is the driver pin. For arcs across interconnect, the sink is the receiver pin.
- 5 3.91 **size metric**: A value used to estimate properties of interconnect wholly contained in a region. The metric may be freely chosen (for example, square microns or gate sites), but it needs to be consistent between the cells and the wireload models. *See also*: **wireload model**.
- 10 3.92 **skew timing check**: A timing check that specifies the maximum time between two signal transitions. This timing check is frequently applied to dual-clock flip-flops to specify the maximum separation of the active edges of the two phases of the clock.
- 3.93 **skip**: In PDEF, spacing between the ordered *cell*, *spare_cell*, and/or *cluster instances* in *rows* and/or *columns* of a *datapath*.
- 15 3.94 **slew**: A measure of the shape of the waveform constituting a logic state transition. A slew value can have the dimensions of time, in which case it is a *slew time*, or the dimensions of voltage-per-time, in which case it is a *slew rate*. The DPCS allows either interpretation if used consistently.
- 20 3.95 **slew-dependent delay**: That part of an input-to-output delay that can be attributed to the signal at the input of the arc taking longer to make a transition than is considered ideal.
- 3.96 **slew rate**: A measure of how quickly a signal takes to make a transition, i.e., a voltage-per-unit time. Slew rate is inversely related to *slew time* and is sometimes used incorrectly where *slew time* is intended.
- 25 3.97 **slew time**: A measure of how long a signal takes to make a transition, i.e., the rise time or fall time. Slew time is inversely related to slew rate. The way a slew time value is abstracted from the continuous waveform at a cell pin varies with different cell characterization methods.
- 30 3.98 **soft region**: A cluster which does not have a specified physical location in a floorplan. It may have constraints on how closely the cells within the cluster are placed relative to each other. A soft region may be located within a hard region.
- 35 3.99 **source, source pin**: The source pin is the start of a delay arc, i.e., the origin of the logic signal. For arcs across cell instances, the source is the receiver pin. For arcs across interconnect, the source is the driver pin.
- 3.100 **spare_cell**: A *cell instance* which is presently not part of the logical function of a design, and therefore is not included in the design's logical netlist. A **spare_cell** is typically reserved for future logic modifications to be implemented through changes in the interconnect layers of the chip.
- 40 3.101 **standard structure**: A particular C structure, defined in `std_stru.h`, which contains fields used to pass data over the PI (thus avoiding large numbers of arguments). Most functions of the PI have a pointer to a *Standard Structure* as their first argument.
- 45 3.102 **technology data**: Data used to calculate the timing properties of a cell instance based on its context in the design. This term includes information that is not cell type-specific and data specific for each cell type in the library. The kind of data used varies with the timing calculation methodology. General data and cell data may be contained in the same file or in separate files. Cell data also may be merged with the timing models of each cell, for example when a tool performs its own timing calculation.
- 50 3.103 **technology library**: A technology library is a program written in DCL consisting of one or more subrules, each of which may contain references to other subrules (yet to be loaded). There is no hierarchical limit to the nesting of subrules within the scope of a technology library. Subrules can also be segmented into technology families, which alters the way they are made available to the application.

3.104 **time-of-flight**: The time delay between a signal leaving a driving pin or primary input port and reaching a receiving pin or primary output or., Time-of-flight is generally dominated by the time taken to charge the distributed capacitance of the interconnect and the capacitance of the driven pins through the distributed impedance of the interconnect. The internal impedance of the driving port affects the load-dependent delay but *not* (directly) the time-of-flight.

3.105 **timing annotation (file)**: The annotation of a design in one tool with timing data computed by another tool. If timing calculation is performed as an off-line process (separately from the application using the timing data), the process of reading the timing data into the tool is known as timing annotation. A timing annotation file stores the data written by the timing calculator and is later read by an application. Syn: **back annotation**.

3.106 **timing arc**: A pair of ports, pins, or nodes possess some timing relationship such as the propagation delay of a signal from one to the other or a timing check between them. Delay arcs may be between two distinct ports or nodes of a cell or over the interconnect from driver pins to receiver pins.

3.107 **timing check**: A timing property of a circuit (frequently a cell) which describes a relationship in time between two input signal events. This relationship needs to be satisfied for the circuit to function correctly.

3.108 **timing model**: A timing model represents the timing behavior of a cell for applications such as simulation and timing analysis. For black-box timing behavior, it represents the definition of pin-to-pin delays between any pair of pins as well as internal nodes. In addition, for sequential cells it provides the definition of timing checks and constraints on any pair of pins and/or internal nodes.

3.109 **transition**: The change of a logic signal from one state to another (as in "... a transition at the input shall cause ...") or the pair of logic states between which a transition may occur (as in "... the delay for a low-to-high transition ...").

3.110 **unloaded delay**: The conceptual delay value for a delay arc of a cell when the output pin is unloaded (unconnected) and the signal at the input pin conforms to some ideal waveform. Syn: **intrinsic delay**.

3.111 **undefined behavior**: Behavior for which the standard imposes no requirements (e.g., use of an erroneous program construct). Permissible undefined behavior ranges from

- ignoring a situation completely with unpredictable results;
- behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message);
- terminating a translation or execution (with the issuance of a diagnostic message).

NOTE—Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed.

3.112 **unspecified behavior**: Behavior (for a correct program construct and correct data) that depends on the implementation. The implementation is not required to document which behavior occurs. Usually the range of possible behaviors is delineated by the standard.

3.113 **via**: In PDEF, a physical connection between two different levels of interconnect, or between a level of interconnect and a physical or logical pin.

3.114 **wireload model**: A statistical model for the estimation of interconnect properties as a function of the geometric measures available before the completion of layout and routing. Typical model properties include: fanout, capacitance, length, and resistance. See also: **size metric**.

4. Acronyms and abbreviations

This section lists the acronyms and abbreviations used in this standard.

API	Application Programming Interface (see also PI)
ASIC	Application Specific Integrated Circuit
AWE	Asymptotic Waveform Evaluation
CAE	Computer-Aided Engineering (the term EDA is preferred)
CFI	CAD Framework Initiative
DCL	Delay Calculation Language
DPCM	Delay and Power Calculation Module
DPCS	Delay and Power Calculation System
D-SPF	Detailed Standard Parasitic Format
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
HDL	Hardware Description Language
OID	Object Identifier
OVI	Open Verilog International
PDEF	Physical Design Exchange Format
PI	Procedural Interface
PVT	Process/Voltage/Temperature
RC	Resistance times Capacitance
RICE	Rapid Interconnect Circuit Evaluator
R-SPF	Reduced Standard Parasitic Format
SDF	Standard Delay Format
Si2	Silicon Integration Initiative
SPEF	Standard Parasitic Exchange Format
SPF	Standard Parasitic Format
SPICE	Simulation Program with Integrated Circuit Emphasis
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VI	VHDL International
VITAL	VHDL Initiative Towards ASIC Libraries
VLSI	Very Large Scale Integration

5. Delay and power calculation system architecture

5.1 Overview

The goals of the Delay and Power Calculation System (DPCS) architecture are to make it possible in a multi-vendor, multi-tool environment, for integrated circuit designers to calculate timing and power *consistently*, for integrated circuit vendors to express timing and power information *once* for a given technology, while enabling sufficient EDA application accuracy.

To meet these goals, the DPCS shall have total control of delay and power calculation, support arbitrary expressions for delay and power values, and have very high performance. In addition, the DPCS architecture shall permit integrated circuit vendors to supply delay and power calculation to their customers independent of the release of design tools by EDA vendors.

Figure 5-1 shows a high-level representation of the DPCS architecture and relationships among several key components for calculating delay and power in an integrated circuit design environment.

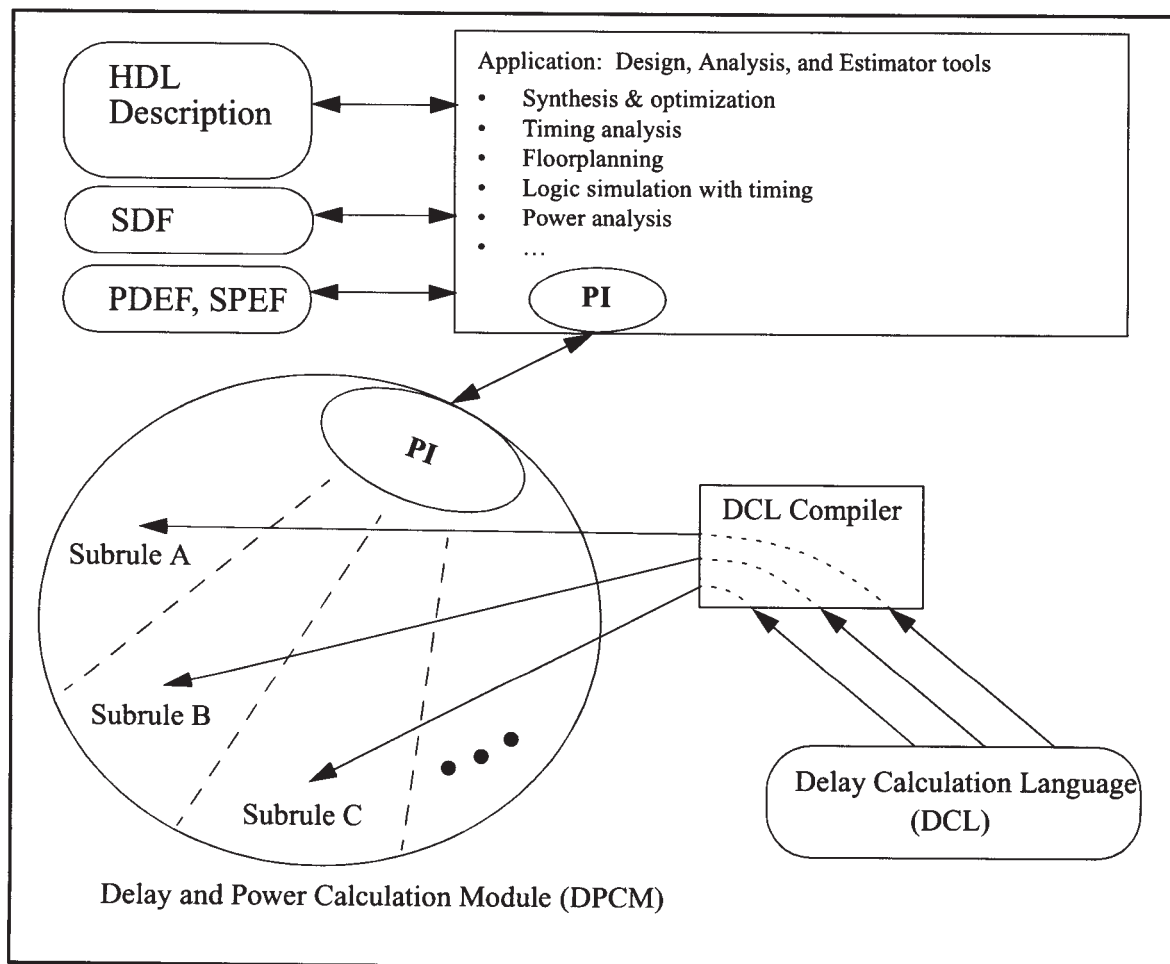


Figure 5-1 High-level DPCS architecture

The Delay and Power Calculation Module (DPCM) contains code (compiled from DCL source) which enables an application to compute power, timing delays and timing constraints efficiently.

- The procedural interface (PI) is used by both an application and the DPCM to control delay and power calculations.
- SDF contains constructs for describing computed timing data (for back-annotation) and specifying timing constraints (for forward-annotation). In integrated circuit design, it is common for a delay calculator application to calculate delays from post-layout data and write out the results in SDF.
- PDEF contains constructs to describe physical design information. It can be used as a back-annotation medium for passing physical design information from back-end applications (floorplanning and layout) to those on the front-end; it can also be used to communicate physical constraints from front-end applications (synthesis, timing analysis, and partitioning) to those on the back-end.
- SPEF contains constructs to convey information between parasitic extractors and applications that utilize the extracted circuit information.

Figure 5-1 also illustrates several key points:

- a) Applications shall be written (or modified) to use the DPCS-defined PI for all delay and power calculations.
- b) The application is fully responsible for mapping between design instances and delay models.
- c) The application is fully responsible for the state of the design. The DPCM knows everything about calculating delays and power, but nothing about a specific design. The DPCM uses the PI to acquire *all* design-specific information, including electrical modeling of parasitic effects and other data (often contained in SDF, PDEF, or SPEF files).
- d) The DPCM is composed of one or more *subrules*. Subrules are compiled C code constructed from an ASCII description following the rules of the Delay Calculation Language (DCL). The DPCM is dynamically linked with the application during execution.
- e) A subrule may implement one or more timing and power models.
- f) One or more timing models may be incrementally patched through the release (and use) of multiple subrules.

5.2 Procedural interface

A crucial part of the DPCS standard is the procedural interface (PI) between an application and the DPCM. This PI enables the same DPCM to function with multiple applications. Details of the PI are described in Clause 8.

To enable an application to be independent of implementation-specific characteristics of a DPCM, code which implements the calls `dcmSetNewStorageManager`, `dcmSetMessageIntercept`, and/or `dcmBindRule` shall not be statically linked with the application. Figure 5-2 illustrates the required linkage structure. The code for these three functions shall reside in a shared object-code library named `libdcmr`, dynamically linked to the application at run-time. There may be implementation-dependent procedure calls between `libdcmr` and the DPCM, but this architecture isolates the application from such details.

NOTE — When the EDA application is statically linked, the link editor shall be told which external symbols need be resolved with dynamically-loaded code and how to locate that code. The specification of how to locate the code shall allow the application to choose from among multiple implementations of `libdcmr`.

5.2.1 Global policies and conventions

The DPCM assumes a single locus of control and data understanding exists for callbacks. For example, if an application passes a pointer to a `port` structure as one argument in a call to the DPCM, the DPCM assumes it can pass that *same* pointer as an argument to a callback function.

In general, the DPCM shall not cache any design-specific data values between calls, since the DPCM cannot know when any such data might become invalid. The DPCM *can* cache within a single application call.

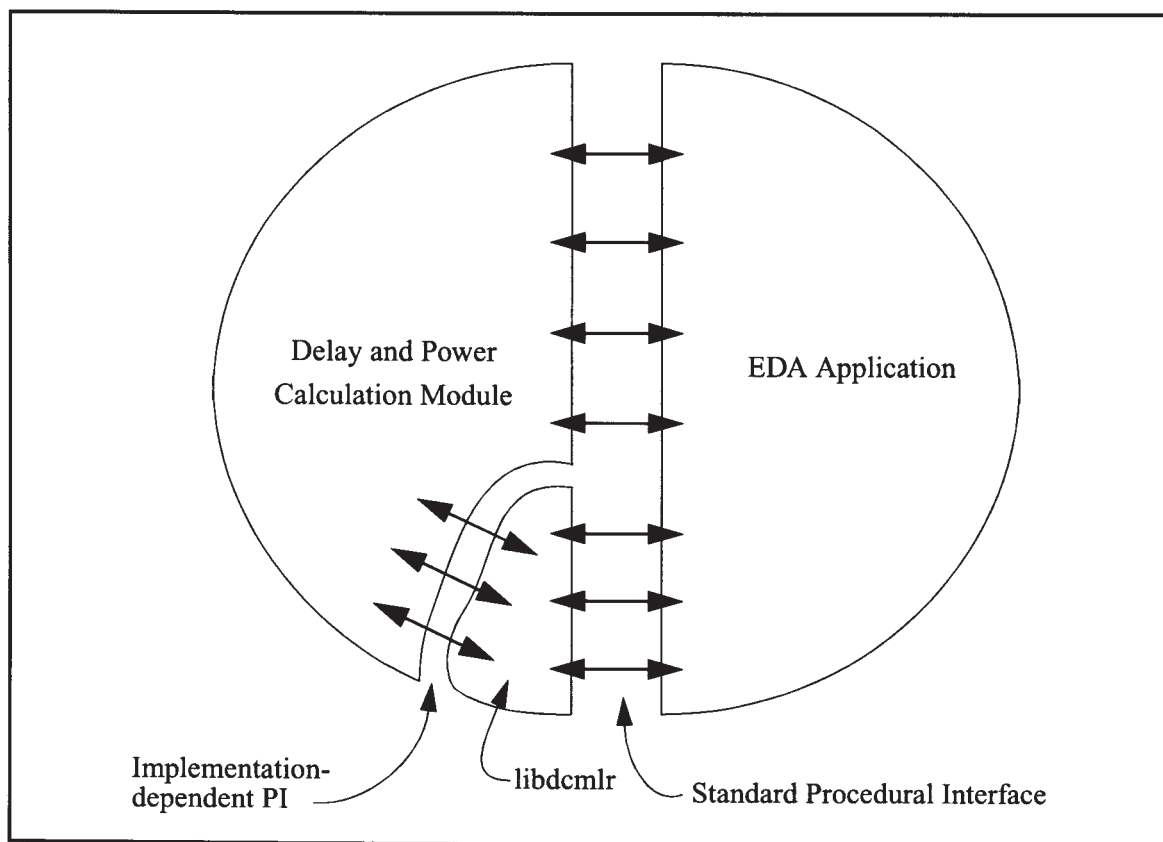


Figure 5-2 High-level DPCS architecture linkage structure

5.2.2 Flow of control

During execution, the flow of control within the DPCS moves between the application and the DPCM, subject to the following two constraints:

- There is no "abnormal" flow of control across its PI, i.e., all function calls return control through the normal procedure-return mechanism.
- A DPCM never terminates the execution of the application to which it is linked.

5.3 DPCM - application relationships

The DPCM has been designed to allow easy integration of dynamically linked executable modules for the purpose of delay and power calculation. The library developer decides how the components of the technology library should be implemented, then organizes the statements into logical groups called technology families and physical units called subrules. The run-time system assembles these units into a unified DPCM. The application then views the DPCM as a single sub-program from which it can request services.

5.3.1 Technology library

A technology library is a program written in DCL consisting of one or more subrules, each of which may contain references to other subrules (yet to be loaded). There is no hierarchical limit to the nesting of subrules within a technology library. Subrules may be segmented into technology families, which alters the way they are made available to the application (see 6.14).

5.3.2 Subrule

A subrule contains a combination of definitions and prototypes used to implement a portion of a technology definition. A subrule is a separate compilation unit and consists of one or more of the following independent components:

- C preprocessor directives
- Embedded C code
- DCL directives
- DCL atomic, model procedure, table manipulation, and library control statements

5.4 Inter-operability

Consideration has been given to the possibility of having multiple implementations of supporting systems for this standard (compiler, run-time linker, and run-time library) and the desire to support at least a minimal level of inter-operability among such implementations.

In particular, an application can concurrently access subrules constructed in multiple implementations by

- a) limiting all subrules in a particular technology family to a single implementation,
- b) appending an implementation-specific suffix to the dcm prefix for all PI calls whose names begin with dcm, and
- c) requiring the application to
 - 1) track which technology families use which implementation,
 - 2) use the appropriately-named PI calls, and
 - 3) maintain separate implementation-specific standard structures.

6. Delay Calculation Language (DCL)

This section describes the delay modeling and calculation used in this standard. The formal syntax is described using the following Backus-Naur Form (BNF). For this clause only following conventions are used:

- a) Lowercase words, some containing embedded underscores, are used to denote syntactic categories (terminals and non-terminals), e.g.,

literal_character_sequence

- b) Boldface words are used to denote reserved keywords, operators, and punctuation marks as a required part of the syntax. These words appear in a larger font for distinction, e.g.,

MODEL SLEW => ;

- c) The **:** **:=** operator separates the two parts of a BNF syntax definition. The syntax category appears to the left of this operator and the syntax description appears to the right of the operator. For example, the following step (d) shows five options for a *timing_mode_operator*.

- d) A vertical bar separates alternative items (use one only) unless it appears in boldface, in which case it stands for itself, e.g.,

timing_mode_operator ::= -> | <- | <-> | <-**X**-> | ->**X**<-

- e) Square brackets enclose optional items unless it appears in boldface, in which case it stands for itself. For example:

result_definition ::= **RESULT** ([result_sequence])

indicates *result_sequence* is an optional syntax item for *result_definition*, whereas

pin_range ::= pin_name [range_expression] pin_name

indicates square brackets are part of the syntax for *pin_range*.

- f) Braces enclose a repeated item unless it appears in boldface, in which case it stands for itself. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

comma_expression_list ::= expression { , expression }

comma_expression_list ::= expression | comma_expression_list , expression

- g) Parenthesis enclose items within a group (use one only) unless it appears in boldface, in which case it stands for itself. In the following example:

bus ::= (**BUS** physical_name { attribute } ({ net_ref } | { pin_ref } { node_ref }))

the first set of parenthesis are part of the syntax for a *bus* and the second set groups the items *net_ref* OR the combination of a *pin_ref* and *node_ref*.

- h) Angle brackets enclose items when no spacing is allowed between the items, such as within an *identifier*. In the following example:

identifier ::= <identifier_first_char>{<identifier_char>}

the actual character(s) of the identifier cannot have any spacing.

- i) A hyphen (-) is used to denote a range. For example:

identifier_first_letter ::= **a-z** | **A-Z**

indicates the first letter of the identifier can be a lowercase letter (from a to z) or an uppercase letter (from A to Z).

- j) If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *msb_constant_expression* and *lsb_constant_expression* are equivalent to *constant_expression*.

The main text uses *italicized* font when a term is being defined, and `monospace` font for examples, file names, and while referring to constants such as 0, 1, or x values.

6.1 Character set

The DCL character set shall be the same as that defined in Section 5.2.1 (“Character sets”) of ISO/IEC 9899:1990, Programming Languages — C, excluding tri-graph sequences and multibyte characters. The alphabetic escape sequences such as ‘\n’, representing newline, shall be the same as those defined in section 5.2.2 (“Character display semantics”) of ISO/IEC 9899:1990, Programming Languages — C

6.2 Lexical elements

This section describes the lexical elements used to define the DCL syntax and semantics.

6.2.1 Whitespace

Whitespace is a contiguous sequence of one or more characters in the set: *space*, *horizontal_tab*, *newline*, *carriage_return*, *vertical_tab*, and *form_feed*.

6.2.2 Comments

A comment shall be either

- a character sequence which starts with `/*` and ends with the first occurrence of the character sequence `*/`. Within a comment, the character sequence `/*` shall not be recognized as starting a nested comment.
- a character sequence which starts with `//` and ends with the first occurrence of either the *newline* or *carriage_return* characters.

6.2.3 Tokens

The syntax for tokens in DCL is given in Syntax 6-1.

```
token ::=
    keyword
    | identifier
    | double_quoted_character_sequence
    | constant
    | string_literal
    | operator
    | punctuation
    | header_name
```

Syntax 6-1—Syntax for DCL token

Tokens other than *string_literal* and *double_quoted_character_sequence* shall not contain embedded whitespace.

6.2.3.1 Keyword

DCL reserves many tokens as *keywords* (see Table 6-1). These keywords shall be used only in the context as defined by DCL. No keyword shall be used as an identifier (see 6.2.3.2).

All keywords shall be case sensitive. Each keyword has two valid syntactic forms: one using only upper-case letters, and one using only lower-case letters. Table 6-1— shows only the upper-case syntactic form.

6.2.3.2 Identifier

An *identifier* is the name which represents a value, except within the PATH, FROM, TO, BUS, TEST, INPUT, NODE and OUTPUT clauses, where it is treated as a literal string. An *identifier* is a sequence of one or more characters starting with an alphabetic character, followed by zero or more alphanumeric or underscore (_) characters.

The syntax for identifiers in DCL is given in Syntax 6-2.

```

identifier ::= <identifier_first_character>{<identifier_character>}
identifier_first_character ::= a-z | A-Z
identifier_character ::= a-z | A-Z | 0-9 | _

```

Syntax 6-2—Syntax for DCL identifier

An *identifier* is case-sensitive. An *identifier* shall not be one of the following:

- a *keyword*
- a reserved word in the C or C++ languages (refer to the ISO C standard)
- any character sequence beginning with the letters DCM in any mixture of case

NOTE — It may be desirable to ensure that some *identifiers* are universally unique, so as to not collide when disparate library pieces from different companies are combined. Examples of such *identifiers* are METHOD names (see 6.12.7) and TECH_FAMILY names (see 6.14.1.1).

6.2.3.3 Double quoted character sequence

A *double_quoted_character_sequence* has the semantics of an *identifier*.

The syntax for double quoted characters in DCL is given in Syntax 6-3.

```

double_quoted_character_sequence ::= " literal_character_sequence "
literal_character_sequence ::= <literal_character>{<literal_character>}
literal_character ::= any character in the ASCII character set except
double_quote, newline, or carriage_return

```

Syntax 6-3—Syntax for DCL double quoted character

1

Table 6-1—Keywords

1	ADD_ROW	FILE	REFERENCE
	ANYIN	FILE_PATH	REFERENCE_EDGE
5	ANYOUT	FILTER	REFERENCE_EDGE_SCALAR
	ASSIGN	FLOAT	REFERENCE_MODE
	BIAS	FORWARD	REFERENCE_MODE_SCALAR
	BLOCK	FROM	REFERENCE_POINT
	BOTH	FROM_POINT	REFERENCE_SLEW
10	BUS	HOLD	REMOVAL
	BY	IMPORT	REPLACE
	CALC	IMPURE	REPEAT
	CALC_MODE	INCONSISTENT	RESULT
	CALC_MODE_SCALAR	INPUT	RISE
15	CALL	INPUT_PIN_COUNT	RULE_PATH
	CELL	INTEGER	SETUP
	CELL_DATA	INTERNAL	SETVAR
	CELL_QUAL	KEY	SIGNAL
	CGHT	LATE	SIGNAL_EDGE
	CGPW	LATE_MODE	SIGNAL_EDGE_SCALAR
20	CGST	LATE_MODE_SCALAR	SIGNAL_MODE
	CHECK	LATE_SLEW	SIGNAL_MODE_SCALAR
	CHECKS	LOAD_TABLE	SIGNAL_POINT
	CKTTYPE	METHOD	SIGNAL_SLEW
	CLKFLG	METHODS	SINK_EDGE
25	COMPARE	MODEL	SINK_EDGE_SCALAR
	COMPILATION_TIME_STAMP	MODEL_DOMAIN	SINK_MODE
	CONSISTENT	MODEL_NAME	SINK_MODE_SCALAR
	CONTROL_PARM	MODELPROC	SKEW
	CPW	MONOLITHIC	SLEW
	CST	NEW	SOURCE_EDGE
30	CYCLEADJ	NIL	SOURCE_EDGE_SCALAR
	DATA	NOCHANGE	SOURCE_MODE
	DEFAULT	NODE	SOURCE_MODE_SCALAR
	DEFINES	NODE_COUNT	STATEMENTS
	DELAY	NUMBER	STORE
35	DELETE_ROW	ONE_TO_Z	STRING
	DHT	OPTIONAL	SUBMODEL
	DO	OTHERWISE	SUBRULE
	DOUBLE	OUTPUT	SUBRULES
	DPW	OUTPUT_PIN_COUNT	SUPPRESS
	DST	OVERRIDE	TABLE
40	DYNAMIC	PASSED	TABLEDEF
	EARLY	PATH	TABLE_PATH
	EARLY_MODE	PATH_DATA	TECH_FAMILY
	EARLY_MODE_SCALAR	PATH_SEPARATOR	TERM
	EARLY_SLEW	PHASE	TEST
45	EDGES	PIN	TEST_TYPE
	END	PINLIST	TO
	EXPORT	PROPAGATE	TO_POINT
	EXPOSE	PROPERTIES	UNLOAD_TABLE
	EXTERNAL	PROTOTYPE_RECORD	UNTIL
50	FALL	PURE	VAR
		QUALIFIERS	VOID
		RECOVERY	WHEN
			Z_TO_ONE
			Z_TO_ZERO
			ZERO_TO_Z

6.2.3.4 Predefined references to Standard Structure fields

DCL defines a set of identifiers to reference fields in the *Standard Structure* (see 8.13). These identifiers shall be visible in all scopes. Table 6-2 lists the predefined identifiers.

Table 6-2—DCL predefined references to Standard Structure fields

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
BLOCK	PIN	Passed in during model search, Passed in calculation.	The instance block identification in the design.
CALC_MODE	STRING	Passed in calculation.	Indicates the type of calculation associated with the current propagate segment calculation: bestCase, worstCase, or nominal.
CALC_MODE_SCALAR	INTEGER	Passed in calculation.	Indicates the type of calculation associated with the current propagate segment calculation: bestCase, worstCase, or nominal.
CELL	STRING	Passed in during model search, Passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL, CELL_QUAL and MODEL_DOMAIN (separated by a period).
CELL_DATA	VOID	Set during model search, Passed in calculation.	Initial pointer to the caching system for cell based store clauses.
CELL_QUAL	STRING	Passed in during model search, Passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL, CELL_QUAL and MODEL_DOMAIN (separated by a period).
CKTTYPE	STRING	Set by DPCM.	Reserved for library developer's use.
CLKFLG	STRING	Set during model search, Passed in calculation.	Holds the clock flag to be assigned to this calculation.
CYCLEADJ	INTEGER	Set during model search.	Reserved for library developer's use.
EARLY_SLEW	FLOAT	Passed in calculation.	Holds the slew associated with the propagate segment for the delay or slew to be calculated. This slew represents the SLEW of the earliest signal to arrive at the segment for which the DELAY or SLEW is being calculated.

Table 6-2—DCL predefined references to Standard Structure fields *(continued)*

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
FROM_POINT	PIN	Set during model search, Passed in calculation.	Holds the from point of the propagate segment for the delay or slew to be calculated.
INPUT_PIN_COUNT	INTEGER	Passed in during model search.	Holds the number of input pins on the circuit currently being modeled or under calculation.
LATE_SLEW	FLOAT	Passed in calculation.	Holds the slew associated with the propagate segment for the delay or slew to be calculated. This represents the SLEW of the latest signal to arrive at the segment for which the DELAY or SLEW is being calculated.
MODEL_DOMAIN	STRING	Passed in during model search, Passed in calculation.	The cell of the circuit under calculation. The unique cell identification is the concatenation of CELL, CELL_QUAL and MODEL_DOMAIN (separated by a period).
MODEL_NAME	STRING	Set during model search.	Holds the name of the modelproc that is currently in control.
NODE_COUNT	INTEGER	Passed in during model search.	Holds the number of nodes connected to the circuit currently being modeled or under calculation.
OUTPUT_PIN_COUNT	INTEGER	Passed in during model search.	Holds the number of output pins connected to the circuit currently under calculation or model build.
PATH	STRING	Set during model search, Passed in calculation.	Holds the path or test statement name.
PATH_DATA	VOID	Set during model search, Passed in calculation.	Holds the pointer to the path and pin base cache
PHASE	STRING	Set by DPCM.	Gives access to phase. This is a combination of the SOURCE_EDGE and SINK_EDGE. When the SOURCE_EDGE and the SINK_EDGE are the same value, PHASE is set to I. If they are not the same value, PHASE is set to O.

Table 6-2—DCL predefined references to Standard Structure fields *(continued)*

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
REFERENCE_EDGE	STRING	Set during model search, Passed in calculation.	Indicates whether the calculation is for the rising, falling, or both edges.
REFERENCE_EDGE_SCALAR	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation is for the rising, falling, or both edges.
REFERENCE_MODE	STRING	Set during model search, Passed in calculation.	Indicates whether the calculation is for early mode, late mode or both.
REFERENCE_MODE_SCALAR	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation is for early mode, late mode or both.
REFERENCE_POINT	PIN	Set during model search, Passed in calculation.	Holds the from point of the propagate segment for the delay or slew to be calculated and is generally used in TEST and TABLEDEF statements.
REFERENCE_SLEW	FLOAT	Passed in calculation.	Holds the slew associated with the test segment for which the check is being analyzed. This slew represents the SLEW of the reference or "clock." This allows the REFERENCE_SLEW to alter the calculations performed by the CHECK statement. The CHECK statement can adjust the BIAS by an amount which is a function of the reference's slew.
SIGNAL_EDGE	STRING	Set during model search, Passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SIGNAL_EDGE_SCALAR	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SIGNAL_MODE	STRING	Set during model search, Passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SIGNAL_MODE_SCALAR	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation represents early, mode, late mode, or both.

Table 6-2—DCL predefined references to Standard Structure fields *(continued)*

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
SIGNAL_POINT	PIN	Set during model search, Passed in calculation.	Holds the to point of the propagate segment for the delay or slew to be calculated and is generally used in TEST and TABLEDEF statements. This is used during test computations.
SIGNAL_SLEW	FLOAT	Passed in calculation.	Holds the slew associated with the test segment for which the check is being analyzed. This slew represents the SLEW of the signal or "data." This allows the SIGNAL_SLEW to alter the CHECK statement's calculation. The BIAS computed by the CHECK statement can be a function of the signal's slew.
SINK_EDGE	STRING	Set during model search, Passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SINK_EDGE_SCALAR	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation represents the rising edge, falling edge, or both.
SINK_MODE	STRING	Set during model search, Passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SINK_MODE_SCALAR	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation represents early mode, late mode, or both.
SOURCE_EDGE	STRING	Set during model search, Passed in calculation.	Indicates whether the calculation is for the rising edge, falling edge, or both.
SOURCE_EDGE_SCALAR	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation is for the rising edge, falling edge, or both.
SOURCE_MODE	STRING	Set during model search, Passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.

Table 6-2—DCL predefined references to Standard Structure fields (continued)

Predefined Identifier	Data Type	Set by DPCM or passed in by application	Description
SOURCE_MODE_SCALAR	INTEGER	Set during model search, Passed in calculation.	Indicates whether the calculation is for early mode, late mode, or both.
TO_POINT	PIN	Set during model search, Passed in calculation.	Holds the to point of the propagate segment for which the delay or slew is to be calculated. It is used during delay and slew computations.

6.2.3.5 Compiler generated predefined identifiers

The DCL compiler generates the values for the predefined identifiers shown in Table 6-3.

Table 6-3—DCL compiler generated predefined identifiers

Predefined Identifier	Data Type	Description
COMPILATION_TIME_STAMP	STRING	The time/date stamp when this rule was compiled.
CONTROL_PARM	STRING	Gives access to the CONTROL_PARM value specified in the SUBRULE statement which loaded this rule.
RULE_PATH	STRING	Gives access to the RULE_PATH value specified in the SUBRULE statement which loaded this rule.
TABLE_PATH	STRING	Gives access to the TABLE_PATH value specified in the SUBRULE statement which loaded this rule.

6.2.3.6 Constant

A *constant* is defined as either a floating point constant number or an integer constant number.

The syntax for constants in DCL is given in Syntax 6-4.

```
constant ::= floating_constant | integer_constant
```

Syntax 6-4—Syntax for DCL constant

The *floating_constant* and *integer_constant* tokens have the same definition as defined in the ISO C standard.

6.2.3.6.1 Pre-defined constant NIL

NIL is a pre-defined constant of ISO C type `void *` with a value of `(void *) 0` which shall indicate the PINLIST, PIN, STRING, ARRAY, or VOID has no value defined.

6.2.3.6.2 Edge type enumerations

The values for the `SINK_EDGE` and `SOURCE_EDGE` predefined identifiers are enumerations of the edge values as shown in Table 6-4. Use of these reserved words results in the corresponding string values in the generated code.

The remaining edge types defined in a `DCM_EdgeTypes` structure: `SAME`, `BOTH`, `ALL`, `COMPLIMENT`, and `TERMINATEBOTH` shall be considered illegal enumeration values for `SINK_EDGE` and `SOURCE_EDGE`.

Table 6-4—Edge types and conversions

Keyword	Definition	String Value	enum Value
FALL	The FALL edge means a signal transitions from a high level to a low level. The levels in DCL are arbitrary as there are no specified reference levels since they are implied in the calculation.	'F'	1
ONE_TO_Z	ONE_TO_Z means the signal is transitioning from a high level to a high impedance state.	'1Z'	7
RISE	The RISE edge means a signal transitions from a low level to a high level. The levels in DCL are arbitrary as there are no specified reference levels since they are implied in the calculation.	'R'	0
TERM	TERM means the edge terminates and does not propagate.	'T'	5
Z_TO_ONE	Z_TO_ONE means the signal is transitioning from a high impedance state to a high level.	'Z1'	8
ZERO_TO_Z	ZERO_TO_Z means the signal is transitioning from a low level to a high impedance state.	'0Z'	9
Z_TO_ZERO	Z_TO_ZERO means the signal is transitioning from a high impedance state to a low level.	'Z0'	10

NOTE — See 8.11.7 for `DCM_EdgeTypes`.

6.2.3.6.3 Propagation type enumerations

`SINK_MODE` and `SOURCE_MODE` are converted to enumerations as shown in Table 6-5.

Table 6-5—Propagation mode conversions

Keyword	Definition	String Value	enum Value
EARLY	The very first edge that propagates through a given cone of logic.	'E'	0
LATE	The very last edge that propagates through a given cone of logic.	'L'	1

6.2.3.6.4 Calculation mode enumeration

CALC_MODE values are converted to enumerations as shown in Table 6-6.

Table 6-6—Calculation mode conversions

Definition	String Value	enum Value
Best case	'B'	0
Worst case	'W'	1
Nominal case	'N'	2

6.2.3.6.5 Test type enumeration

TEST_TYPE values are converted enumerations as shown in Table 6-7.

Table 6-7—TEST_TYPE conversions

Keyword	Definition	enum Value
CGHT	Clock Gating Hold test	7
CGPW	Clock Gating Pulse Width test	6
CGST	Clock Gating Setup test	8
CPW	Clock Pulse Width test	2
CST	Clock Separation test	3
DHT	Data Hold Test	10
DPW	Data Pulse Width test	4
DST	Data Setup test	5
HOLD	Hold test	1
NOCHANGE	No change test	14
RECOVERY	Recovery test	11
REMOVAL	Removal test	12
SETUP	Setup test	0
SKEW	Skew test	13

6.2.3.7 String literal

A *literal_character* is any member of the character set except the single-quote ('), backslash (\), or *newline_character*. Each of these restricted characters may be present in a single quoted string if it is preceded by a backslash.

The syntax for string literals in DCL is given in Syntax 6-5.

```
string_literal ::= ' {< literal_character> }'
```

Syntax 6-5—Syntax for DCL string literal

string_literals follow the same semantics as ISO C string literals (see the ISO C standard, Section 6.1.4).

6.2.3.8 Operators

The syntax for DCL operators is given in Syntax 6-6.

```
operator ::=
    $ | ^ | * | / | + | = | ** | || | && | ! | == | != | > | < | >= | <=
    | -> | <- | <-> | ->X<- | <-X-> | NEW | :: | .
```

Syntax 6-6—Syntax for DCL operator

6.2.3.9 Punctuator

The syntax for punctuators is given in Syntax 6-7.

```
punctuator ::= ( ) | [ ] | { } | , | ; | : | . | # | & | < | > | % { } % ' | "
```

Syntax 6-7—Syntax for DCL punctuator

The punctuators `()`, `[]`, `%{ }%`, and `{ }` shall occur in balanced pairs.

6.2.3.10 Name

A *name* is either an *identifier* or a *double_quoted_character_sequence*. The use of these punctuators is incorporated within the subsequent syntax charts in this clause.

6.2.4 Header names

Header name preprocessing tokens shall only appear with a `#include` preprocessing directive. The header name shall be defined the same as in section 6.1.7 ("Header Names") of the ISO C standard.

6.2.5 Preprocessing directives

DCL preprocessing directives are exactly the set defined in Section 6.8 of the ISO C standard and have the same semantics.

6.3 Name spaces of identifiers

If more than one declaration of a particular identifier is visible at any point in the translation unit, the syntactic context determines proper reference to different entities. Thus, there are separate name spaces for various categories of identifiers, as follows:

- identifiers declared in RESULT clauses,
- identifiers naming methods and technology families, and
- all other identifiers.

6.4 Storage durations of objects

An object has a *storage duration* which determines its lifetime. There are two storage durations: *static* and *automatic*.

An object defined by an ASSIGN statement or whose identifier is declared with external or internal linkage shall have *static storage duration*. For such an object, storage shall be reserved prior to program start-up. The object shall exist and retain its last-stored value throughout the execution of the entire program.

An object defined by a STORE clause shall have *static storage duration*. For such an object, storage shall be reserved when the enclosing MODELPROC statement is evaluated. The object shall exist and retain its last-stored value throughout the execution of the entire program.

All non-array objects whose identifiers are not defined by a STORE clause or an ASSIGN statement shall have *automatic storage duration*. For such objects, their storage shall be allocated when the defining statement is called and shall persist until the calling statement goes out of scope.

For all array objects, their storage shall be allocated by the NEW operator and shall persist (while in scope within the DPCM) at least until control reaches the application or while the application or the DPCM has locked the array (see 8.4.1).

6.5 Scope of identifiers

An *identifier* shall be visible (i.e., can be used) only within a region of program text referred to as its *scope*. There are ten different types of scope:

- global
- TECH_FAMILY
- subrule
- statement-prototype
- statement
- modeling procedure
- STATEMENTS
- RESULT
- PASSED
- discrete

When lexically identical identifiers exist in the same name space, identifiers in outer scopes shall be hidden until the inner scope terminates. The DCL scope hierarchy is:

```
global
    TECH_FAMILY
```

```

1          subrule
              statement-prototype
              statement
                  RESULT
5                      PASSED
                          discrete
              modeling procedure
                  STATEMENTS
                      statement
10

```

The scope of an *identifier* shall start where the *identifier* is first declared and extend to the end of the scope in which it was declared. An *identifier* shall be declared before it is referenced. Multiple MODELPROC statements of the same name shall not occur within the same TECH_FAMILY.

6.6 Linkages of identifiers

A statement *identifier* declared in different scopes can be made to refer to the same object or statement by a process called *linkage*. There are three subrule scope options affecting linkage: EXPORT, IMPORT, and FORWARD. Expose chaining also defines how EXPOSE identifiers are linked within a TECH_FAMILY's scope. (See 8.8.1.1 for more details.)

6.6.1 EXPORT

A statement *identifier* shall be made visible outside its declared subrule scope and within the same TECH_FAMILY by using the EXPORT option on a statement definition.

6.6.2 IMPORT

A statement *identifier* EXPORTed in one subrule scope shall be made visible within another subrule scope and within the same TECH_FAMILY by using the IMPORT option on a statement prototype.

6.6.3 FORWARD

A statement *identifier* referenced in a subrule scope before its definition (within the same scope) shall be made visible by using the FORWARD option on a statement prototype.

6.6.4 Chaining of EXPOSE identifiers

EXPOSE statement identifiers with the same name in separate subrules within the same TECH_FAMILY are chained together at run-time. The application and DPCM are presented with a single EXPOSE entry point for this identifier. The first EXPOSE statement in a chain is always used as the reference to any EXPOSE statement within a particular subrule (even if there is an EXPOSE statement defined within the subrule containing the reference).

6.7 DCL data types

DCL supports three categories of data types: *native*, *derived*, and *array*. The native data types are defined in terms of the ISO C data types. See 8.11.6 for details of the mapping between DCL and ISO C data types.

6.7.1 Native data types

The syntax for native data types in DCL is given in Syntax 6-8.

```
native_type ::= mathematical_type | pointer_data_type
```

Syntax 6-8—Syntax for DCL native data type

6.7.1.1 Mathematical calculation data types

The syntax for calculation data types in DCL is given in Syntax 6-9.

```
mathematical_type ::= NUMBER | DOUBLE | FLOAT | INTEGER
```

Syntax 6-9—Syntax for DCL calculation data type

6.7.1.2 Pointer data types

The syntax for pointer data types in DCL is given in Syntax 6-10.

```
pointer_data_type ::= STRING | PIN | PINLIST | VOID
```

Syntax 6-10—Syntax for DCL pointer data type

6.7.2 Array types

The syntax for array types in DCL is given in Syntax 6-11.

```
native_array_type ::= [ mathematical_type | STRING | PIN | VOID ] [
    dimension_list ]
dimension_list ::= fixed_dimension_list | variable_dimension_list
fixed_dimension_list ::= scalar { , scalar }
variable_dimension_list ::= * { , * }
```

Syntax 6-11—Syntax for DCL array type

An array type describes a contiguously allocated nonempty set of objects with a particular member object type, called the *element type*. Array types are modeled by their element type, their number of dimensions, and the number of elements in each dimension of the array.

An array dimension may have type *fixed* or *variable*. The number of elements in a *fixed* dimension shall be specified when the array is declared and shall not vary during program execution. The number of elements in a *variable* dimension shall not be specified when the array is declared and may vary during program execution.

The following are true for all array types:

- Arrays can have an arbitrary number of dimensions up to a limit of 255. For any particular array, all of its dimensions shall have the same type — fixed or variable.
- The length of each dimension in a multi-dimensional array shall not depend on the index used for any of the other dimensions of that array — “ragged” arrays shall not be supported.
- Array indices shall start with zero (0) in each dimension.
- Array types shall be passed and returned as pointers to a contiguous memory region. The array data shall be organized as defined by the ISO C standard.

Array types shall be constant unless declared to be VAR. A VAR array may have its elements changed during program execution. An array type that has the VAR declaration may be passed as an argument to a statement requiring a constant array. A constant array type shall not be passed as an argument to a statement requiring a VAR array type.

The VAR array modifier is defined in Table 6-8.

Table 6-8—VAR array modifier

Symbol	Definition
VAR, var	Indicates (to the compiler) the array field is modifiable by any variable that references the array. Without the var array modifier, arrays are constant and cannot be modified in any way.

Example

```
FORWARD CALC(name) :
  RESULT(INTEGER[*,*]: constIntArray &
    NUMBER VAR [*]: varNumVec);
```

The VAR array modifier indicates array elements may be altered by statements other than the statement that created the array. Assignments for non-var to non-var, var to var, or var to non-var are effected through pointer manipulation; no copying of array values is performed.

6.7.3 Derived data types

The syntax for derived data types is given in Syntax 6-12.

1

5

10

```

derived_type ::= result_type | statement_type
result_type ::=
    assign_statement_name
    | calc_statement_name
    | expose_statement_name
    | external_statement_name
    | internal_statement_name
    | tabledef_statement_name
statement_type ::= result_type ( )

```

15

Syntax 6-12—Syntax for DCL derived data type

6.7.3.1 Result types

20

RESULT clauses in statements and DATA clauses in TABLEDEFs define derived data types which shall be analogous to structures in the ISO C standard. The name of the associated statement or TABLE can be used as the name of the derived data type. RESULT type field ordering is defined section 6.12.1.2.1 and 6.12.1.2.2.

Example

25

30

```

calc(resultType) :    passed(integer:x)
                      result(double:fp=2*x+3 & integer:x+1);
calc(example) :      passed(resultType:x)
                      result(double:x.fp*4 - 2*x);
calc(exercise) :     result(double:example(resultType(5)));

```

6.7.3.2 Statement types

35

A statement name followed by () defines a derived data type. Any statement having the same PASSED arguments (number, order, and type) and RESULT structure matches this derived type. This enables the passing of statement pointers into other statements.

Example

40

45

50

```

exercise() passes statementType1() to example() as a derived type:

calc(statementType1) :  passed(integer:x)
                       result(integer:resType(x) + 3*x);
calc(statementType2) :  passed(integer:z)
                       result(integer:resType(z)+6*z+ 1);
calc(example) :         passed(statementType1(): y)
                       result(integer: y(5));
calc(exercise) :        result(integer:example(statementType1)+
                               example(statementType2));

```

6.8 Type conversions

Type conversions shall be performed based on

- changing the type of an argument to the type of the expected parameter or
- changing the type of an expression term to the type of other terms in the same expression.

Only the conversions enumerated in the following subsections are valid in DCL.

6.8.1 Implicit conversions

The following implicit type conversions shall be performed:

- INTEGER to DOUBLE or FLOAT
An INTEGER value shall be converted to DOUBLE (FLOAT) when the expected parameter or expression term has type DOUBLE (FLOAT). An INTEGER value shall be converted to DOUBLE when a division operator is present in the expression.
- FLOAT to DOUBLE
A FLOAT value shall be converted to DOUBLE when the expression term has type DOUBLE.
- PIN to STRING
A PIN value shall be converted to STRING when the expected parameter or expression term has type STRING.
- native_array_type, pointer_data_type or derived_type to VOID
A native_array_type, pointer_data_type or derived_type value shall be converted to VOID when the expected parameter or expression term has type VOID.

6.8.2 Explicit conversions

The following explicit type conversions shall be performed:

- FIXED-dimension array to VARIABLE-dimension array
A FIXED-dimension array shall be converted to a VARIABLE-dimension array when a fixed dimension array is assigned to a variable dimension array of the same dimensionality and data type.
- DERIVED data type to DERIVED data type
A DERIVED data type shall be converted to an equivalent DERIVED data type when the expected parameter has the same structure (order, number, and native types) as the argument.

Example:

```

CALC(resType): passed(integer:x)
                result(double:fp = 2*x+3 & integer:x+1);
CALC(resType2):passed(integer:x)
                result(double:y=2**x & integer:z=x-1);
CALC(example): passed(resType:x)
                result(double:x.fp * 4 - 2 * x);
CALC(exercise):result(double:example(resType(5))+example(resType2(3)));

```

6.9 Operators

This section details the operators used in DCL.

6.9.1 String prefix operator

This subsection details the use of string prefix operators.

6.9.1.1 Explicit string prefix operator

The *explicit string prefix operator* is a unary operator which modifies the semantics of logical comparisons between string operands. This operator can be used with string operands within logical expressions.

The explicit string prefix operator is the * (asterisk) character. This operator shall occur as a separate token which precedes a string literal or an *identifier* having a string value.

6.9.1.2 Embedded string prefix operator

The *embedded string prefix operator* is a unary operator which modifies the semantics of logical comparisons between string operands. This operator can be used with string operands within logical expressions, table qualifiers, and table references.

The embedded string prefix operator is the * (asterisk) character. This operator shall occur as the first character of the string literal, the first character of a name, or the first character of the value of an identifier. To disable this special meaning of an asterisk, it shall be preceded with a backslash character (\).

6.9.1.3 String prefix semantics

The semantics of the explicit and embedded string prefix operator are identical. When the string prefix operator is present, the logical comparison shall be performed considering only a subset of the characters in the strings. The comparison shall start with the first character after the prefix operator, if present, in each string and shall proceed for the length of the string with the string prefix operator. If both strings have the prefix operator, then the comparison shall proceed for the length of the shorter string.

If the explicit string prefix operator is used and its operand contains a leading asterisk, this leading asterisk shall *not* be considered an operator; it shall be treated as the first character of the string.

6.9.2 Assignment operator

For native types, except arrays, the assignment operator (=) shall assign the expression value on the right to a variable on the left. For arrays, this operator shall make the variable on the left (which shall be an array name) a *reference* to the array on the right.

6.9.3 New operator

The new operator (NEW) shall be used to create memory space for arrays. NEW is used when a new instance of an array is needed. After the NEW operator creates the space for an array, the values of the individual members shall be undefined.

6.9.4 SCOPE operator

The scoping prefix PATH_DATA : : or CELL_DATA : : determines what *action statement* is associated with a method (see 6.12.7), and what the value of a store variable is, based on the value of the corresponding pre-defined identifier PATH_DATA or CELL_DATA (see Syntax 6-16).

6.9.5 Purity operator

A reference to an impure statement shall be treated as PURE if the reference is preceded by the purity operator (^) (see 6.12.2.1). The purity operator is defined in Table 6-9.

Table 6-9—Purity operator

Symbol	Definition
^	The unary purity operator declares the statement reference immediately following it to be PURE, thereby overriding any default assumptions based on the optimization rules. References prefixed with the purity operator shall be optimized as a PURE reference.

6.9.6 Timing propagation

Static timing includes the need to resolve timing when two or more input signals converge at a node. For each signal, a time period (window) can be defined based on the earliest and latest possible times when that signal can arrive at the node. The resolution process at a node determines a window for the node's output signal based on the windows of the input signals for that node.

DCL supports five ways (modes) to perform this resolution, defined in "Timing resolution modes" on page 56 and used in PROPAGATE, EDGE and TEST clauses.

Table 6-10—Timing resolution modes

Symbol	Definition
->	early(output)= earliest of all early(input) late(output)= early(output)
<-	late(output)= latest of all late(input) early(output)= late(output)
<->	early(output)= earliest of all early(input) late(output)= latest of all late(input)
->X<-	early(output)= latest of all early(input) late(output)= latest of all late(input)
<-X->	early(output)= earliest of all early(input) late(output)= earliest of all late(input)

6.9.7 Timing checks

Static timing includes the need to compare the windows of different signals at different nodes. One signal is always chosen as the reference. The comparison done is determined by a combination of test typeSee "Test type enumeration" on page 47., test mode operator See "Test mode operators" on page 57., and EDGE clause.

DCL supports three ways to perform this comparison, defined in "Test mode operators" on page 57.

Table 6-11—Test mode operators

Symbol	Definition
->	signal must arrive later than reference + bias Bias shall be computed using early signal and late reference values.
<-	signal must arrive earlier than reference - bias Bias shall be computed using late signal and early reference values.
<->	signal must arrive earlier than reference - offset1, and signal must arrive later than reference + offset2

There are four possible window-edge comparisons - the early or late edge of the signal window can be compared with the early or late edge of the reference window. The test mode operand refines the window comparison by specifying which ends of the windows to compare, along with some additional semantics.

Typically, two windows are maintained for each signal, corresponding to rising and falling transitions. The EDGE clause selects which window is used for the comparison.

6.9.8 Test mode operators

This subsection details the use of test mode operators.

6.9.8.1 CGHT

The Clock Gating Hold test mode operator (CGHT) is similar to the HOLD test mode operator except the reference edge shall be from a clock pin and the signal shall be from a data pin.

6.9.8.2 CGPW

The Clock Gating Pulse Width test mode operator (CGPW) is similar to the CPW mode operator except the pin specified for the test shall be from a clock gate (logic signal).

6.9.8.3 CGST

The Clock Gating Setup test mode operator (CGST) is similar to the SETUP test mode operator except the reference edge shall be from a data pin and the signal shall be from a clock pin.

6.9.8.4 CPW

The Clock Pulse Width test mode operator (CPW) specifies the edge identified as the signal shall be offset from the edge identified by the reference. The amount of the offset is at least as large as the bias value. The edge identified as the signal shall occur before the edge identified as the reference for late mode and positive bias values. The edge identified as the signal shall arrive after the edge identified as the reference for early mode and positive bias values. Both edges specified for this test shall be from the same pin and a clock.

6.9.8.5 CST

The Clock Separation test mode operator (CST) specifies the edge identified as the signal shall be offset from the edge identified as the reference. The amount of the offset is at least as large as the bias value. The