General Constraint Format Specification

Version 1.2

August 22, 1997

Cadence Design Systems, Inc.

Contents

1

Introduction

Introduction

The General Constraint Format (GCF) file is intended to be used for interchanging constraint data associated with a design between EDA tools used at any stage in the design process. The data in the GCF file is represented in a tool-independent way and can currently include

- Timing environment: intended operating timing environment
- Timing constraints
- Parasitics constraints
- Parasitics environment: intended operating parasitics environment
- Area constraints
- Power constraints
- Design/instance-specific or type/library-specific data

Cadence Design Systems expects that other types of constraint data will be added to the GCF specification in the future, such as

- Analog constraints
- Noise and signal integrity constraints

A particular GCF file can contain all of these types of constraints, or it can contain only certain types of constraints.

GCF is not intended to represent detailed constraints such as the timing checks described in the Standard Delay Format (SDF), as SDF is already well-defined for this information. Instead, GCF covers many types of constraints for which no standard currently exists.

The name of each GCF file is determined by the EDA tool. There are no conventions for naming GCF files.

Published by Cadence Design Systems

Cadence Design Systems has developed this GCF specification to enable accurate and unambiguous transfer of constraint data between tools that require this information. ***All parties utilizing the GCF should interpret and manipulate constraint data according to this specification.*** Please direct questions and corrections to:

Mark Hahn
Cadence Design Systems
3945 Freedom Circle, 4th Floor
Santa Clara, CA 95054

Tel: (408) 450-6548
Fax: (408) 748-3499
internet e-mail: mhahn@cadence.com

Acknowledgements

The Constraint Forum working group of Cadence Design Systems acknowledges the individual and team efforts invested in establishing this version of the GCF specification:

Mark Hahn (primary author)

Ria Simons-Arnout (editor)

Suzanne Thomas (editor)

Henry Chang

Edoardo Charbon

James Cherry

Geoffrey Ellis

Theo Kelessoglou

Anandi Krishnamurthy

Enrico Malavasi

Ed Martinage

Dave Noice

Sherry Solden

Ted Vucurevich

The SDF 3.0 specification developed by Open Verilog International has strongly influenced GCF. The organization and format of the GCF document and the contents of a number of sections are borrowed loosely from SDF. The intent is to build upon this excellent previous work as a foundation for a broader description of the designer's intent, particularly with respect to timing.

Version History

Version 1.2 - August 22, 1997

- Modified the semantics of the **DEPARTURE_TIME** construct to directly correspond to setup and hold times of a virtual register connected to the output.
- Added an **EXTERNAL_DELAY** construct which describes purely combinational delays external to a cell.
- Modified the **PATH_DELAY** construct semantics to reflect the **EXTERNAL_DELAY** construct, and to allow cell instances and waveform names to be specified as endpoints.
- Added a section on default precedence rules, as well as a number of specific precedence rules for particular constraints and sets of constraints.

Version 1.1 - July 8, 1997

- Added internal slew and clock slew constructs.
- Modified the **CLOCK_DELAY** construct to allow the leaf pins to be omitted, in which case all primitive clock input pins reachable from the specified root are implied.
- Modified the **PATH_DELAY** construct to allow each of the rise min, rise max, fall min, fall max delays to be specified independently.
- Updated the **DRIVER_CELL**, **DRIVER_STRENGTH**, and **INPUT_SLEW** constructs to explicitly state that if no *port_instance* is specified, then the construct applies by default to all primary input and bidirectional pins.
- Fixed conflicting statements about whether the **ARRIVAL** and **DEPARTURE** constructs allow internal pins to be specified as well as primary i/o's. The statements have been corrected to indicate that internal pins are allowed.
- Added the '<' and '>' characters as legal bus delimiters.
- Added the syntax description for *disable_cell_spec_1*, which was missing in Version 1.0.
- Fixed minor inconsistencies.
- Extensive editing to improve readability.

Version 1.0 - March 21, 1997

- Added operating conditions and voltage thresholds to the environment globals. Added the ability to override the operating conditions for part of the design in Level 1.

- Changed the semantics of the process, voltage, and temperature constructs to specify the range of operating conditions over which the design is intended to operate.
- Modified the default voltage thresholds to be 10% and 90% instead of 20% and 80%.
- Added a restriction on clock waveforms to only allow a single pair of edges.
- Added an *r_rise_fall_min_max* value type, which allows for negative arrival and departure times, and an INUMBER variable, which represents a possibly negative integer.
- Dropped the delay offset construct.
- Moved fanout-based parasitics constructs to Level 1, since these require wire load models to interpret.
- Updated the driver cell construct to allow distinguishing between the cell types which should be used for each type of edge.
- Modified the CLOCK_TREE construct and renamed it to CLOCK_DELAY.
- Modified name prefixes to include the number of prefixes, and to require that the id numbers be sequential starting at 0.
- Modified the max transition time check to refer to output pins, rather than load pins.
- Significantly modified the disables section to eliminate problems with overloading several different types of disables into a single syntax.

Version 0.7 - January 24, 1997

- Significantly expanded the description of the multi-cycle constraint semantics and modified them to better match existing tools.
- Modified the syntax to allow Level 1 constraints to be grouped together within a GCF section.
- Fixed many minor inconsistencies between different sections of the document.

Version 0.6 - November 15, 1996

- Added many new kinds of information:
  - Case-dependent constraints
  - Constant signal specifications
  - Clock domains
  - Process, voltage, and temperature specifications
  - Area and power constraints.
  - Meta data describing the precedence between alternate constraints.

- Significantly revised many of the timing constraints to better match the semantics of existing tools.
- Separated constraints into several levels of support.
- Modified the syntax to reduce verboseness and eliminate ambiguities when using yacc as the basis for parsing.

Version 0.5 - April 15, 1996

- Incorporated feedback from internal review.

Version 0.4 - April 8, 1996

- Initial formal version for internal review.

2

GCF in the Design Process

GCF in the Design Process

Sharing of Constraint Data

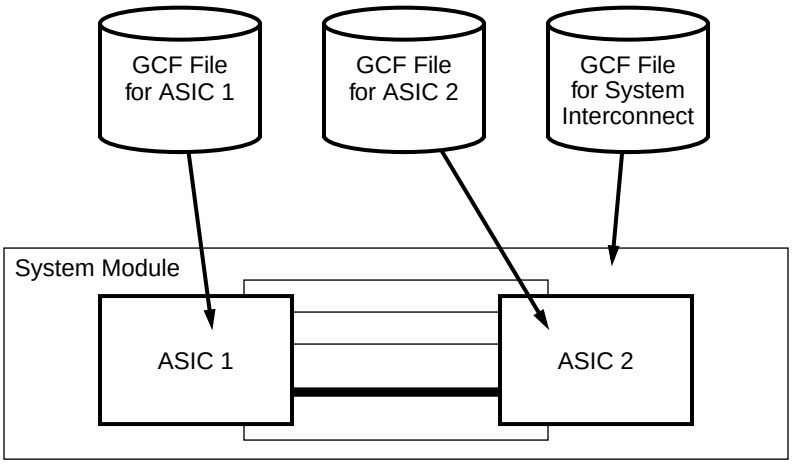
By accessing a GCF file, EDA tools are assured of consistent, accurate, and up-to-date data. This means that EDA tools can use data created by other tools as input to their own processes. By sharing data in this way, estimation, synthesis, floorplanning, analysis, and layout tools can all use a consistent set of design constraints with well-defined semantics.

The EDA tools create, read (to update their design), and write to GCF files.

Using Multiple GCF Files in One Design

GCF files support hierarchical constraint annotation. A design hierarchy might include several different ASICs (and/or cells or blocks within ASICs), each with its own GCF file as illustrated in Figure 1.

Figure 1 Multiple GCF Files in a Hierarchical Design



Timing Environment

GCF includes constructs for describing the intended timing environment in which a design will operate. For example, you can specify the waveform to be applied at clock inputs and the arrival time of primary inputs.

Some of the timing environment information is also covered by SDF 3.0. You should use SDF to pass delay data and detailed path constraints between tools and use GCF to pass high-level timing constraints and the timing environment description between tools.

GCF contains a richer description of the environment, particularly in terms of the information required for doing delay calculation on interface nets. It also supports many types of timing constraints which are not covered by SDF.

Timing Constraints

GCF contains constructs for describing special cases within a sequential circuit, such as false and multi-cycle paths. It also contains constructs

which allow constraints to be applied on combinational or asynchronous parts of a circuit.

Parasitic Constraints

GCF contains constructs for describing constraints on the parasitics within a circuit, such as a limit on the internal capacitance of interface nets. These constraints would typically be used by synthesis and layout tools.

Parasitic Environment

GCF includes constructs for describing the parasitics in the environment in which a design will operate. For example, you can specify the external capacitance for interface nets.

Area Constraints

GCF contains constructs for constraining the primitive area and the total area of a cell, as well as the porosity of the cell.

Power Constraints

GCF includes constructs for constraining the average power consumed by a cell and the average power dissipated by the capacitance in a net.

The GCF Creator

One or more tools can be responsible for generating the GCF file. For example, a synthesis tool or a dedicated constraint management tool can capture constraint information from the designer and then write out this data in GCF. To do this, it will examine the specific design for which it has been instructed to generate constraint data. Tools which create GCF files must locate, within the design, each region for which constraint data exists and calculate values for the parameters of those constraints.

Many types of constraints, such as clock waveform descriptions, apply throughout the design process. Other types of constraints, such as parasitic constraints on an interconnection, can be derived from high-level timing constraints. GCF supports describing both high-level and derived constraints in the same file. Thus, GCF is suitable for both prelayout and postlayout applications.

There are provisions in the GCF specification for adding meta data associated with constraints in a later revision. This meta data can be used in many ways; some planned uses include describing relationships between constraints, and describing the relative importance of each constraint. The meta data will refer to constraints through a unique *label* which can be associated with each constraint.

Many tools only need a description of the constraints themselves, and do not require any of the meta data. However, tools which create GCF files should not make assumptions about the requirements of the tools which

will read the GCF file. To prevent the need for multiple GCF files with different sets of meta data for a given design, a tool which creates GCF files should include as much meta data as possible. Each reader is expected to filter out the meta data it does not require. Tools which create GCF files can make judicious use of the *include* construct to make this filtering efficient.

GCF imposes no restrictions on the precision which is used to represent the data in a GCF file. Therefore, the accuracy of the data in the GCF file will depend on the accuracy of the constraint generator and the information made available to it.

The Annotator

The GCF file is brought into a reader tool through an annotator. The job of the annotator is to match data in the GCF file with the design description. Each region in the design identified in the GCF file must be located. Constraints in the GCF file for this region must be applied to the appropriate parameters of the design.

The annotator can be instructed to apply the data in the GCF file to a specific region of the design, other than at the top level of the design hierarchy. In this case, it will search for regions identified in the GCF file starting at this point in the hierarchy. The file must clearly have been prepared with this in mind, otherwise the annotator will be unable to match what it finds in the file with the design viewed from this point.

The foregoing implies that the annotator must have access to the design description. Frequently, this will be via the internal representations maintained by the reader tool. The annotator will then be a part of the tool. As an alternative, the annotator can operate independently of the reader tool and convert the data in the GCF file into a format suitable for the tool to read directly. If such an annotator is unable to match the GCF file to the design description, then the effect of inconsistencies is unpredictable. Also, certain constructs of GCF cannot be supported without access to the design description (for example, wildcard cell instance specifications and wildcard bit specifications).

Consistency Between GCF File and Design Description

A GCF file contains constraint data for a specific design. The contents of the file identifies regions of the design and provides constraints that apply to various properties of that region. The analysis tool or annotator cannot operate if the regions identified in the GCF file do not correspond exactly with the design description. Therefore, changes to the design sometimes require writing a new GCF file, depending on the types of changes and constraints. A future version of GCF might provide a mechanism for describing incremental changes to an existing GCF file.

Of equal importance to the logic of the design is the naming of design objects. Even if the same cells are present and are connected in the same way, annotation cannot operate if the names by which these cells and nets

are known differ in the GCF file and the design description. The naming of objects must be consistent in these two places.

During annotation, inconsistencies between the GCF file and the design description are considered errors.

Consistency Between GCF File and Analysis

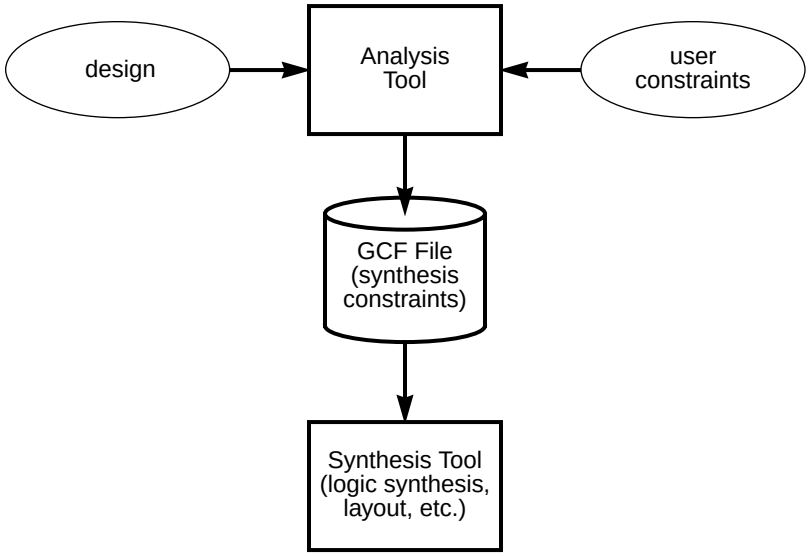
GCF includes a description of a standard semantics for many kinds of constraints. Some tools might not support all of the types of constraints in GCF, or might restrict the semantics for some types of constraints. For example, a layout tool might handle disabling of false paths where a single port is specified, but not handle disabling of false paths where multiple ports are specified.

The constraints of GCF are divided into a number of subsets, where each subset contains constraints associated with a particular aspect of a circuit, such as timing or parasitics. When a tool reads a GCF file, it can choose to read one or more of these subsets. During the annotation of each subset a tool reads, unsupported constraints or unsupported semantics for a constraint are considered to be warnings. However, a tool should not warn about unsupported constraints in other subsets.

Forward-Annotation of Constraints for Design Synthesis

In addition to the use of constraint data for analysis and estimation, GCF supports the forward-annotation of constraints to design synthesis tools. (In this context, we use the term “synthesis” in its broad sense of construction, thus including not only logic synthesis, but also floorplanning, layout and routing.) Constraints are “requirements” for the design’s overall properties and are often modified and broken down by previous steps in the design process. Figure 2 shows a typical scenario of the use of GCF in a design synthesis environment.

Figure 2 GCF Files in Constraint Forward-Annotation



Constraints can also be originated by an analysis tool alone. For example, a timing budgeting tool might be able to propagate the high-level timing constraints specified by a designer down to each hierarchical module in the design, setting arrival time and departure time constraints on each module port automatically.

3

Using GCF

GCF File Content

GCF files are ASCII text files. Every GCF file contains a header section followed by one or more additional sections. A GCF file can contain zero cell entries.

Syntax

constraint_file ::= **(GCF** *header section+* **)**

section ::= *globals*
||= *cell_spec*
||= *extension*
||= *meta_data*
||= *include*

The *header* section contains information relevant to the entire file such as the design name, the tool used to generate the GCF file, and scaling factors for the values in the file (see “Header Section” on page 28).

The *globals* section describes information which is common to all cells in a design.

Each cell construct, *cell_spec*, identifies part of the design (a “region” or “scope”) and contains data for the constraints on that part of the design (see “Cell Entries” on page 58). A *cell* can be a physical primitive from the ASIC library, a modeling primitive for a specific analysis tool or some user-created part of the design hierarchy. A *cell* can encompass the entire design.

Extensions provide a mechanism to extend the standard GCF format with user-defined portions.

Meta data describes relationships between constraints.

This chapter describes the header, globals, cell-spec, and a number of GCF-specific concepts (such as levels, cases, labels, include files, value types, and design references). The following chapters describe specific subsets in GCF. For each part of the file, the purpose is discussed, the syntax is specified, and an example is presented. A complete, formal definition of the file syntax is contained in Chapter 6, “Syntax of GCF.” You can refer to that chapter for precise definitions of some of the abbreviated syntax descriptions given here.

Header Section

The header section of a GCF file contains information that relates to the file as a whole. Except for the GCF version, entries are optional, so that it is possible to omit most of the header section.

The design name, date, and program entries are for documentation purposes and do not affect the meaning of the data in the rest of the file. However, the version, delimiters, and scaling factors do affect how the data in the file is interpreted.

Syntax

header ::= **(HEADER** *version header_info** **)**

header_info ::= *design_name*
||= *date*
||= *program*
||= *delimiters*
||= *time_scale*
||= *cap_scale*
||= *res_scale*
||= *length_scale*
||= *area_scale*
||= *voltage_scale*
||= *power_scale*
||= *extension*

GCF Version

The version construct identifies the version of the GCF specification to which the file conforms.

Syntax

version ::= **(VERSION** QSTRING **)**

QSTRING is a character string in double quotes. The first substring within QSTRING, which consists of just numeric characters and a period, identifies the GCF version. Other characters before and after this substring are permitted and will be ignored by readers when determining the GCF version.

Example

```
(VERSION “Cadence Version 1.2”)
```

Readers of GCF files can use the GCF version construct to adapt to the differences in file syntax between versions. If the file does not contain a GCF version construct, or one is present but the QSTRING field does not contain a numeric substring, the GCF reader will give an error message.

Design Name

The design name construct specifies the name of the design to which the constraints in the GCF file apply. This construct is for documentation purposes only.

Syntax

design_name ::= **(DESIGN** QSTRING **)**

QSTRING is a name that identifies the design. Although this construct is not used by the annotator, it is recommended that, if it is included, the name should be the name given to the top level of the design description. This is analogous to the **CELLTYPE** construct, and in fact, the same name would be used in a cell construct for the entire design. It must not be the instance name of the design in a test-bench; this would instead be used as part of the cell instance path in the **INSTANCE** entries for all cells.

Date

The date construct indicates how current the data in the file is. This construct is for documentation purposes only.

Syntax

date ::= **(DATE** QSTRING **)**

The QSTRING represents the date or time when the data in the GCF file was generated or last modified.

Example

```
(DATE “Friday, June 6, 1997 - 7:30 p.m.”)
```

Program

The program name construct indicates the name of the program that created or last modified the file. This construct is for documentation purposes only.

Syntax

program ::= **(PROGRAM**
program_name program_version
program_company **)**

program_name ::= QSTRING

program_version ::= QSTRING

program_company ::= QSTRING

The QSTRING parameters contain (respectively)

- The name of the program used to generate or modify the GCF file
- The version number of that program
- The company that produced the program

Example

```
(PROGRAM “GCF writer” “1.1” “Cadence”)
```

Delimiters

The delimiters construct specifies the characters that are used as delimiters in design names.

Syntax

delimiters ::= **(DELIMITERS** QSTRING **)**

The QSTRING always contains three characters:

- The first character is referred to as the hierarchy delimiter character, or HCHAR, and must be either a period (.) or a slash (/). If there is no *delimiters* construct in the GCF file, the HCHAR defaults to a period.
- The second character is referred to as the left index character, or LI_CHAR, and must be either a left bracket ([), a left parenthesis ((), or a left angle bracket (<). If there is no *delimiters* construct in the GCF file, the LI_CHAR defaults to a left bracket.
- The third character is referred to as the right index character, or RI_CHAR, and must be either a right bracket (]), a right parenthesis ()), or a right angle bracket (>). If there is no *delimiters* construct in the GCF file, the RI_CHAR defaults to a right bracket.

Example

```
  (DELIMITERS “/()”)
. . .
   (INSTANCE a/b/c(3))
. . .
```

In this example, the hierarchy delimiter is specified to be the slash (/) character, so the hierarchical paths use the slash (rather than the period) to separate elements. In addition, the left and right index characters are set to be parentheses, so that bit-specs for selecting elements from instance arrays or buses are specified using parentheses (rather than brackets).

Hierarchical delimiters can be used in an IDENTIFIER and a PATH. Index characters can be used in an IDENTIFIER. For more information, see “Variables” on page 119.

Scaling Factors

A scaling factor entry specifies the multiplier to be used to scale the values for the specified physical property.

Syntax

time_scale ::= **(TIME_SCALE** *multiplier* **)**
cap_scale ::= **(CAP_SCALE** *multiplier* **)**
res_scale ::= **(RES_SCALE** *multiplier* **)**
length_scale ::= **(LENGTH_SCALE** *multiplier* **)**
area_scale ::= **(AREA_SCALE** *multiplier* **)**
voltage_scale ::= **(VOLTAGE_SCALE** *multiplier* **)**
power_scale ::= **(POWER_SCALE** *multiplier* **)**
multiplier ::= NUMBER

The default time scale is 1 second. If *time_scale* is specified, the GCF reader will multiply all delay numbers in the GCF file by the specified value, which is in seconds. For example, a multiplier of 1.0E-12 corresponds to delay values in ps.

The default capacitance scale is 1 Farad. If *cap_scale* is specified, the GCF reader will multiply all capacitance numbers in the GCF file by the specified value, which is in Farads. For example, a multiplier of 1.0E-12 corresponds to capacitance values in pF.

The default resistance scale is 1 ohm. If *res_scale* is specified, the GCF reader will multiply all resistance numbers in the GCF file by the specified value, which is in ohms. For example, a multiplier of 1.0E-3 corresponds to resistance values in milli-ohms.

The default length scale is 1 meter. If *length_scale* is specified, the GCF reader will multiply all length numbers in the GCF file by the specified value, which is in meters. For example, a multiplier of 1.0E-6 corresponds to length values in microns.

The default area scale is 1 square meter. If *area_scale* is specified, the GCF reader will multiply all area numbers in the GCF file by the specified value, which is in square meters. For example, a multiplier of 1.0E-12 corresponds to area values in square microns.

The default voltage scale is 1 volt. If *voltage_scale* is specified, the GCF reader will multiply all voltage numbers in the GCF file by the specified value, which is in volts. For example, a multiplier of 1.0E-3 corresponds to voltage values in millivolts.

The default power scale is 1 watt. If *power_scale* is specified, the GCF reader will multiply all power numbers in the GCF file by the specified value, which is in watts. For example, a multiplier of 1.0E-3 corresponds to power values in milliwatts.

Example

```
(CAP_SCALE 1.0E-12)
```

Levels

GCF provides a mechanism for interchanging constraint data between many different kinds of tools. The capabilities of each tool affect the types of constraints that the tool can support.

It is desirable to standardize as many types of constraints as possible to ensure that the tools that support each constraint do so in a consistent way. However, this presents a dilemma to a designer who is using GCF: What constraints can be used successfully given the set of tools that the designer must use?

GCF divides the constraints into several levels of support. In this version of GCF, two levels have been identified. In this document, all constraints are Level 0 unless otherwise specified.

Level 0

Level 0 provides a baseline capability to which most tools will conform. It includes the most important basic constraints. These constraints are widely supported already, and the algorithms required to support the constraints are well understood and relatively straightforward to implement.

A designer or a flow developer might choose to use only the Level 0 constraints so that the GCF file is widely portable across different tools.

Tool vendors should state whether their tools comply with Level 0 on a subset-by-subset basis. For example, a timing analysis tool vendor might state that the tool fully supports GCF Level 0 (Timing and Parasitics subsets).

Level 1

Level 1 includes additional constraints that are less widely supported but are viewed as important for certain design styles or methodologies. These constraints generally allow a more precise description of the intended operation of the circuit than can be expressed using just the Level 0 constraints.

Level 1 constraints might require more complex algorithms which affect the performance of a tool. On the other hand, a tool might achieve better quality results or perform a more accurate analysis when Level 1 constraints are used.

A designer or a flow developer can choose to use some or all of the Level 1 constraints. This decision is necessarily more difficult than choosing to use only Level 0 constraints. It requires careful analysis of at least the following:

- The performance versus accuracy trade-off
- The tools that support the desired Level 1 constraints
- The resulting effect if not all of the tools support all of the constraints

Even when some aspect of the design behavior can't be expressed properly by using Level 0 constraints, it is likely that a designer still needs to specify Level 0 constraints (which are overly restrictive) so that tools which only support Level 0 can produce correct results.

In a flow that mixes tools supporting Level 0 and Level 1 constraints, it is desirable to specify the Level 1 constraints as well. If both constraints are specified in the same GCF file, it is ambiguous which constraints will be used by a Level 1 tool. In this case, the **PRECEDENCE** construct can be used to describe the relationship between the constraints (see "Meta Data" on page 40).

Usage

It is desirable that every tool can read a GCF file containing both Level 0 and Level 1 constraints, so that a single GCF can be used throughout a flow. The syntax for GCF has been defined in a way that allows tools which only support Level 0 to easily ignore Level 1.

Level 0 constraints are not explicitly identified as belonging to Level 0, while Level 1 and higher constraints must appear within the *level* construct.

The general form for the level construct is shown below. There are a number of variations of the level construct, where each variation restricts the types of level-specific constraints which can appear at a particular point in the GCF file.

Syntax

level ::= **(LEVEL** NUMBER *construct+* **)**

A precise description of each type of level specification is included in Chapter 6, "Syntax of GCF."

For this version of GCF, NUMBER must be set to 1.

Cases

With some design styles, it is either necessary or convenient to separate the constraints into several different cases. For example, you can use cases

- To distinguish between major modes of operation (such as, normal mode versus test mode and reset mode)
- To describe the circuit behavior when several clocks are muxed together
- To describe the effect of gating clocks

Some tools do not support case-dependent constraints, some tools handle each case separately without considering the interactions between them, and some tools can look at each case separately, as well as consider the interactions between them.

Because not all tools support case-dependent constraints, these constraints are included in GCF Level 1, but not in Level 0. However, given that there are a number of tools which do support case analysis, there is value in being able to describe the cases in a consistent way.

Cases are identified in GCF using a unique identifier. Unless they appear within the *case* construct, all constraints in a GCF belong to the *default* case. The name *default* cannot be used to identify other cases.

The general form for case specifications is shown below. The description of a case-dependent constraint depends on the context in which it is used.

Syntax

case_spec ::= **(CASE** IDENTIFIER
case_dependent_constraint+ **)**

Each case is likely to be described using a number of different *case_spec* constructs in different places in the GCF. The unique identifier for the case must be used in each of the *case_spec* constructs associated with the case.

A precise description of each type of case specification is included in Chapter 6, “GCF File Syntax.”

Constant Values

In addition to allowing constraints to be separated into different cases, GCF also allows specifying that certain signals have a constant value in a given case. In this respect, case-dependent constraints are similar to state-dependent delays. However, state-dependent delays are commonly expressed using Boolean expressions on signal values. In GCF, there is an implicit AND of the constant values specified for a given state.

Constant specifications appear within the timing subset for the cell which contains the *port_instance* (see “Timing Environment” on page 66).

Extensions

There are a number of cases in which it is desirable to extend a standard format such as GCF in unofficial ways:

- For preliminary testing of official proposals for new versions of the format
- For early versions of evolving portions of the format
- For representing company-specific, flow-specific, or tool-specific data which is not suitable for standardization but is strongly related to the data in the standard (Often, a separate data format is appropriate for these cases, but in some cases having a separate data format would require duplicating much of the information)

However, there are also several concerns with unofficial extensions:

- Unofficial extensions might be used indefinitely for data that should become part of the official standard.
- Without a built-in mechanism for extensions, most GCF readers would not be able to read a GCF file containing an extension. This would greatly limit the use of extensions because all of the readers in a particular design flow would have to be modified for each extension. With a built-in mechanism for extensions, only tools requiring the data included in the extension would need to be modified.

To overcome the latter concern, GCF includes a built-in mechanism for unofficial extensions, and establishes a policy restricting the syntax of those extensions.

We expect that there will be periodic revisions to GCF to incorporate additional types of constraint data. The developers of unofficial extensions to GCF are strongly encouraged to submit their extensions for standardization; this makes the data in the extensions more widely accessible and promotes greater tool interoperability.

Syntax

extension ::= **(** **EXTENSION** QSTRING *extension_construct+* **)**

extension_construct ::= **(** *user_defined* **)**
||= *include*

The QSTRING contains the name of the extension. Extension names must be unique. For example, an extension name might include the name of the tools which support it.

Extensions must conform to the GCF syntax for parenthesized constructs and strings to enable every GCF reader to ignore the extension by searching for a matching right parenthesis that is not embedded within a quoted string.

Except for these restrictions, the format for the extension is flexible. Any keywords can be used, including existing GCF keywords. There is no limit on the number of the parenthesized constructs associated with an extension, and extension constructs can be arbitrarily nested.

Extensions must not be inserted at arbitrary points in a GCF file. They can only be included where explicit provisions were made in the GCF syntax.

Example

```
(EXTENSION “color”
  (PACKAGE_COLOR “white” “grey” “black” )
)
```

In this example, an extension is defined for a constraint on the possible colors of the package containing the design, where the color must be one of the listed values.

Precedence Rules

Some types of constraints can be expressed in several similar forms. Each of these forms results in different degrees of accuracy. Ideally, only the most accurate form would be included in the GCF, and all tools would support this form.

For example, the effect of an external driver on delay calculation for an interface signal can be described by identifying the cell and its drive strength or by specifying an input slew. Identifying the cell is the most accurate approach in most cases.

Unfortunately, not all the tools in a given flow support the same forms of a constraint. In this case, it isn't possible to create a single GCF file with only one form of a constraint and go through the flow successfully.

GCF allows multiple forms of a constraint to be included in a single GCF file. For tools that only support one form of the constraint, there isn't any question about what the tool will do. But for tools that support several forms of the constraint, a set of default precedence rules are defined in order to make it clear which form will be applied. There is also a capability in Level 1 to explicitly override the default precedence rules.

Normal Precedence Rules

In the absence of any explicit precedence overrides, the following general precedence rules are used. Specific precedence rules are also given for particular constructs and sets of constructs in the section of the specification which describes those constructs.

- A value which is given explicitly for a particular port instance or cell instance always overrides a default value. Another way to say this is that the default value only applies to design elements for which a value was not explicitly specified.
- If two different values are given explicitly for the same port instance or cell instance, the value which appears later in the GCF file is used.
- If two different default values are given, the default value which appears later in the GCF file is used.

Overrides

See "Meta Data" on page 40 for a description of how to explicitly override the default precedence rules.

Meta Data

This version of GCF primarily describes basic constraint data. Meta data is information about the relationships between constraints or about how to apply the constraints. Meta data is only supported in Level 1.

One form of meta data is included in this version of GCF, and there are explicit provisions for adding other forms of meta data in the future. The goal is to avoid having to change existing GCF readers as more meta data is added unless a reader chooses to support the meta data.

Precedence Overrides

The supported form of meta data describes the precedence among several related constraints. The precedence meta data construct allows the user to explicitly override the default precedence for a set of several constraints. A tool that supports the precedence meta data applies just one constraint from the set. The chosen constraint will be the highest precedence constraint which the tool supports; the remaining constraints in the set are ignored.

Other Meta Data

There are many other types of meta data which might be added to GCF in future versions. For example, tools often convert constraints of one type into constraints of another type. The meta data might include a description of the transformation algorithm which should be used or the parameters used in the transformation.

Another example is constraint propagation (decomposing high-level constraints on a design into lower-level constraints on each portion of the design). The meta data might include a description of the dependency between the high-level constraint and the lower-level constraints.

Often it is not strictly necessary to satisfy every individual constraint. It might be acceptable to make trade-offs between different constraints. Failing to meet a particular constraint might not be catastrophic.

For example, capacitance constraints can be budgeted for each net in a design. Even though a number of nets fail to meet their constraints, the circuit can still function properly if other nets more than satisfy their constraint. Meta data could describe which constraints must be strictly satisfied (such as the cycle time) and which constraints are only goals that help to ensure that the strict constraints are satisfied.

A designer often sets constraints on a number of different aspects of a circuit, such as area, timing, and power. If not all of these constraints can

be satisfied, the designer can use meta data to describe the relative importance of each aspect.

Usage

Meta data usually must refer to constraints. To allow constraint references, the constraints must be uniquely labeled. For more information, see “Labels” on page 43.

Syntax

meta_data ::= **(LEVEL 1** *meta_data_1+* **)**

meta_data_1 ::= **(META** *meta_construct+* **)**

meta_construct ::= *precedence*
||= *meta_reserved*
||= *include*

precedence ::= **(PRECEDENCE (** *label_id label_id+* **))**

meta_reserved ::= **(** IDENTIFIER ***reserved_for_future_definition*** **)**

Constraints must be listed in the **PRECEDENCE** construct in decreasing order of precedence: the first label in the list is the most preferred constraint.

The IDENTIFIER is used to distinguish between other types of meta data; explicit values for this will be established in future revisions to GCF.

Example

```
(META (PRECEDENCE (label1 label2)))
```

This example describes the precedence between two different constraints identified as *label1* and *label2*. The description of these constraints must precede the **META** construct in the GCF file. If a tool supports the constraint referenced by *label1*, it will apply that constraint. Otherwise, if it supports the constraint referenced by *label2*, it will apply that constraint. If it doesn’t support either constraint, the tool will give a warning.

Include Files

GCF is intended to be the basis for describing a broad range of different types of constraints of varying levels of detail, as well as meta data associated with those constraints. Therefore, it is likely that a complete GCF file for a design will be fairly large.

The GCF syntax organizes related data by cell type, subsets, extensions, and meta data. By creating separate files for each cell type, subset, extension, or type of meta data, a GCF writer can make it as efficient as possible for reader applications to find and read just the relevant data. This has to be weighed against the cost of reading from multiple files and the additional complexity for the user of maintaining multiple files.

Every GCF reader must accept the specification of a search path containing a list of directories in which to search for included files when a relative file name is specified. The user interface for this specification is not defined by GCF. Two common approaches are the -I command line options used by many compilers and the PATH environment variable used by the shell in UNIX. Supporting a similar interface is recommended for UNIX-based GCF readers.

If a file is not found in any of the directories listed in the search path, the GCF reader will give an error message.

Syntax

include ::= **(INCLUDE** QSTRING **)**

The QSTRING specifies the name of the file to be included. GCF writers will use relative file names to allow a set of GCF files to be copied from one location to another. Relative file names are interpreted with respect to the file that contains the include specification, not with respect to the current working directory of a reader.

The GCF syntax describes explicitly where the include construct can be used. An include file which is referenced at a particular point in the GCF must contain only data that would, if substituted directly at that point, conform to the GCF specification. The intent of these restrictions is to make it possible for a reader application to easily identify those include files which it does not have to read at all because they can only contain data that is not relevant to the reader.

Labels

Labels can be used to identify constraints within a GCF file. Consequently, each label within a GCF file must be unique. The label must be an identifier or a quoted string if the label is a GCF keyword.

There is a provision for a label in every basic constraint construct of GCF.

Syntax

label ::= *label_id* COLON

label_id ::= IDENTIFIER
||= QSTRING

A simple and compact approach for a GCF writer is to assign consecutive integers as labels. If desired, more information can be conveyed in the label by using a quoted string.

If several GCF writers are used to create different subsets, a policy must be established to ensure that the labels created by the writers do not conflict. An example policy would be to include an abbreviation for the subset name at the start of the label, such as TG0, TG1, ... for labels within the timing globals subset.

Example

```
(27: INTERNAL_LOAD 10.0 out6)
```

In this example, the label is 27, and it uniquely identifies a constraint on the internal load of the net connected to pin *out6*.

Value Types

Most constraints take one or more values, and there are similar restrictions on the types of values which are legal. This section describes a number of basic value types which are used in other constructs.

Syntax

```
min_and_max ::= NUMBER NUMBER
r_min_and_max ::= RNUMBER RNUMBER
min_max ::= NUMBER NUMBER?
r_min_max ::= RNUMBER RNUMBER?
rise_fall_min_max ::= NUMBER
                  ||= NUMBER NUMBER
                  ||= NUMBER NUMBER NUMBER NUMBER
r_rise_fall_min_max ::= RNUMBER
                    ||= RNUMBER RNUMBER
                    ||= RNUMBER RNUMBER
                            RNUMBER RNUMBER
rise_and_fall ::= NUMBER NUMBER
r_rise_and_fall ::= RNUMBER RNUMBER
rise_fall ::= NUMBER NUMBER?
r_rise_fall ::= RNUMBER RNUMBER?
```

The formal definitions of NUMBER and RNUMBER can be found under “Variables” on page 119.

Globals

The globals section describes the constraint data that applies to multiple cells within the design. Use of the globals section avoids duplication of constraint data within each cell. The globals section must appear before any *cell_spec* sections.

Syntax

globals ::= **(GLOBALS** *globals_subset+* **)**

globals_subset ::= *env_globals_subset*
||= *timing_globals_subset*
||= *extension*
||= *meta_data*

This version of the GCF defines two types of global data: the environment globals subset and the timing globals subset.

Environment Globals

The environment globals subset describes the operating conditions for a design, including process, temperature, and voltage values. There are two types of specifications: a range specification, which describes the range of values over which the design is intended to operate, and a corner specification, which describes a particular process, voltage, and temperature point for which analysis or optimization is to be done.

The environment globals subset also describes the voltage thresholds used for the slew specifications and maximum transition constraints in other parts of the GCF.

In Level 1, the operating conditions can be case-dependent.

Syntax

env_globals_subset ::= **(GLOBALS_SUBSET ENVIRONMENT**
env_globals_body **)**

env_globals_body ::= *env_globals_spec+*
||= *include*

env_globals_spec ::= *env_globals_spec_0*
||= *env_globals_spec_1*

env_globals_spec_0 ::= *process*
||= *voltage*
||= *temperature*
||= *operating_conditions*
||= *voltage_threshold*
||= *extension*
||= *meta_data*

env_globals_spec_1 ::= **(LEVEL** 1 *env_globals_case+* **)**

env_globals_case ::= **(CASE** IDENTIFIER *env_globals_spec_0+* **)**

Example

```
(GLOBALS_SUBSET ENVIRONMENT
  (voltage 4.5 5.5)
  (operating_conditions “fastest” 0.8 3.1 -25.0)
)
```

In this example, only the voltage range is specified, and the process corner to be used for analysis corresponds to the fastest delays.

Process

The *process* construct specifies the range of process derating factors over which the design is intended to operate. This range restricts the *process_value* which can be specified for the operating conditions.

Syntax

process ::= **(** *label?* **PROCESS** *min_and_max* **)**

Example

```
(process 0.8 1.2)
```

In this example, assuming that 1.0 represents a nominal process, the process derating factor used for analysis can vary by plus or minus 20 percent.

Voltage

The *voltage* construct specifies the range of voltages over the design is intended to operate. This range restricts the *voltage_value* which can be specified for the operating conditions.

Syntax

voltage ::= **(** *label?* **VOLTAGE** *r_min_and_max* **)**

The *r_min_and_max* parameter specifies minimum and maximum voltages.

Example

```
(voltage 2.9 3.1)
```

In this example, assuming that the voltage scaling factor is set to 1.0, the design is intended to operate with a supply voltage between 2.9 and 3.1 volts.

Temperature

The *temperature* construct specifies the range of temperatures over which the design is intended to operate. This range restricts the *temperature_value* which can be specified for the operating conditions.

Syntax

temperature ::= **(** *label?* **TEMPERATURE** *r_min_and_max* **)**

The *r_min_and_max* parameter specifies the minimum and maximum operating ambient temperatures in degrees Celsius (centigrade).

Example

```
(temperature -25.0 85.0)
```

In this example, the design is intended to operate between -25.0 and 85.0 degrees Celsius.

Operating Conditions

The *operating_conditions* construct specifies an environmental corner—a particular combination of process, voltage, and temperature derating points —for which analysis or optimization is to be done.

Syntax

operating_conditions ::= **(** *label?* **OPERATING_CONDITIONS**
QSTRING
process_value
voltage_value
temperature_value)

process_value ::= NUMBER

voltage_value ::= RNUMBER

temperature_value ::= RNUMBER

The QSTRING parameter specifies a name for the environment corner, which is used in some libraries to obtain the models for converting the process, voltage, and temperature derating points into delay multipliers.

The *process_value* specifies the process derating point. The interpretation and the units of the derating factor are library-dependent. The process derating point is used to compute a multiplier for scaling delays to reflect the impact of variations in the process. Usually the derating point is interpreted as an index into a linear model which defines the delay multiplier.

If the GFC file contains a *process* construct that defines a range of allowable process derating points, the *process_value* must fall within that range. There is no default range.

The *voltage_value* specifies the voltage derating point, which has units specified by the *voltage_scale*. The voltage derating point is used to compute a multiplier for scaling delays to reflect the impact of variations in the supply voltage. Usually the derating point is interpreted as an index into a linear model which defines the delay multiplier.

If the GFC file contains a *voltage* construct that defines a range of allowable voltages, the *voltage_value* must fall within that range. There is no default range.

The *temperature_value* specifies the temperature derating point in degrees Celsius (centigrade). The temperature derating point is used to compute a multiplier for scaling delays to reflect the impact of variations in the ambient temperature. Usually the derating point is interpreted as an index into a linear model which defines the delay multiplier.

If the GFC file contains a *temperature* construct that defines a range of allowable temperatures, the operating *temperature_value* must fall within that range. There is no default range.

The operating conditions defined in the global environment subset apply by default to all cells in the design. In Level 1, this can be overridden for particular cells by including an *operating_conditions* specification in the timing subset for a cell.

Example

```
(operating_conditions “slowest” 1.2 2.9 85.0)
```

In this example, the environment corner is set to reflect derating points which result in the analysis or optimization being based on the slowest delays.

Voltage Threshold

The *voltage_threshold* construct specifies the measurement points on a waveform which must be used in calculating a slew or transition time. The measurement points are defined as a fraction of the change in voltage from the start of the transition to the end of the transition. If no voltage thresholds are specified in a GCF file, the default values are 10% and 90%.

Syntax

voltage_threshold ::= **(** *label?* **VOLTAGE_THRESHOLD** *min_and_max* **)**

The *min_and_max* parameter specifies the minimum and maximum measurement points.

Example

```
(voltage_threshold 20.0 80.0)
```

In this example, the measurement points on the waveform are at the 20% and 80% points with respect to the change in voltage associated with the transition.

Environment Globals Case

The environment globals can be case-dependent.

Syntax

env_globals_spec_1 ::= **(LEVEL 1** *env_globals_1+* **)**

env_globals_1 ::= *env_globals_case*

env_globals_case ::= **(CASE** IDENTIFIER *env_globals_case_spec+* **)**

env_globals_case_spec ::= *env_globals_spec_0*

Example

```
(GLOBALS_SUBSET ENVIRONMENT
  (level 1
    (case board1
      (voltage 4.5 5.5)
    )
    (case board2
      (voltage 3.1 3.5)
    )
  )
)
```

In this example, the voltage range depends on the board in which the design is used.

Timing Globals

The timing globals subset defines waveforms, derived waveforms, and clock domains. Waveforms and their derivatives can be referenced by each cell, as needed. A clock domain is a group of clocks which are synchronous with respect to each other.

Syntax

timing_globals_subset ::= **(GLOBALS_SUBSET TIMING** *timing_globals_body* **)**

timing_globals_body ::= *timing_globals_spec+*
||= *include*

timing_globals_spec ::= *timing_globals_spec_0*
||= *timing_globals_spec_1*

timing_globals_spec_0 ::= *primary_waveform*
||= *extension*
||= *meta_data*

timing_globals_spec_1 ::= **(LEVEL** 1 *timing_globals_1+* **)**

timing_globals_1 ::= *timing_globals_no_case_1*
||= *timing_globals_case*

timing_globals_no_case_1::= *derived_waveform*
||= *clock_group*

The following sections describe primary waveforms, derived waveforms, clock groups, and timing globals case.

Example

```
(GLOBALS_SUBSET TIMING
  (include “global_timing.gcf”)
)
```

In this example, the global timing constraints are described in a separate file, global_timing.gcf, which must located in a directory along the search path.

Primary Waveform

The primary waveform construct defines an abstract periodic waveform, which is not necessarily associated with any particular signal in the portion of the design described by the GCF file. A waveform typically is used to define one or more clock signals.

The following example uses a waveform that isn’t associated with any signal. The GCF file for a chip might need to refer to the waveform of an off-chip clock in a constraint on the arrival time at an input pin of the chip, but that clock itself might not be supplied to the chip.

The primary and derived waveform constructs allow multiple pairs of edges. However, when a waveform description is used to define a clock or

is used as a reference for an arrival or departure time, the waveform must only have a single pair of edges.

Syntax

primary_waveform ::= **(** *label?* **WAVEFORM** *waveform_name* *period edge_pair_list* **)**

waveform_name ::= QSTRING

period ::= NUMBER

edge_pair_list ::= *pos_pair+*
||= *neg_pair+*

pos_pair ::= *pos_edge neg_edge*

neg_pair ::= *neg_edge pos_edge*

pos_edge ::= **(POSEDGE** *min_max* **)**

neg_edge ::= **(NEGEDGE** *min_max* **)**

The name of the waveform must be unique. The period describes the interval at which the waveform repeats, and is in units of time.

All waveforms are described with respect to an implicit reference point in time. When a circuit contains several clock domains (see “Clock Groups” on page 53), there is one implicit reference point for each clock domain which applies to all of the clocks in that domain. The clock waveforms within a clock domain must be described relative to the implicit reference point, so that known skew between related clocks is reflected in the respective waveform edge positions.

There is no relationship between the reference points for different clock domains.

edge_pair_list describes a single period of the waveform. It consists of a list of edge pairs, which can be either a *pos_edge* construct followed by a *neg_edge* construct or a *neg_edge* construct followed by a *pos_edge* construct. Thus, the total number of edges in the list will be even and the edges will alternate between **POSEDGE** and **NEGEDGE**.

In addition to the direction of the transition, each edge gives the time at which the transition takes place relative to the start of each period. Offsets must increase monotonically throughout the *edge_pair_list* and must not exceed the period.

The *min_max* entries allow either one or two values to be specified for each edge. If one value is given, then this precisely defines the transition offset. If two values are given, then they define an uncertainty region in which the transition will take place. This would usually be used to describe jitter in a clock signal. The first value gives the beginning of the uncertainty region and the second value gives its end. Tools using this construct with two values will assume that a single transition of the specified direction occurs somewhere in the uncertainty region but can not make any assumptions

about the exact location. Tools unable to model this edge uncertainty will issue a warning message and use the mean of the two values to locate the transition.

Example

```
(WAVEFORM “100 MHz 50/50” 10.0
  (POSEDGE 0) (NEGEDGE 5.0)
)
```

In this example, a waveform is defined with a 50% duty cycle and a 10 ns period (assuming that the time_scale construct specifies that delay values in the file are in ns).

Derived Waveform

The derived waveform construct defines a waveform that is harmonically related to a previously defined waveform (the “parent” waveform which might itself be a derived waveform). Derived waveforms can only be specified in Level 1.

Derived waveforms are commonly used in a multi-phase, single-frequency clocked system. A single abstract waveform is defined, and other phases are derived from it.

Another example of when this is useful is when clock multipliers or dividers are used to convert one clock waveform into another waveform with a different but related frequency. By defining the output waveform of a divider as a derived waveform, a change to the definition of the period of the parent waveform will automatically affect the output waveform.

Syntax

derived_waveform ::= **(** *label?* **DERIVED_WAVEFORM**
waveform_name
parent_waveform_name
period_multiplier? phase_shift?
skew_adjustment? **)**

parent_waveform_name ::= QSTRING

period_multiplier ::= **(PERIOD_MULTIPLIER** DNUMBER **)**

phase_shift ::= **(PHASE_SHIFT** RNUMBER **)**

skew_adjustment ::= **(SKEW_ADJUSTMENT** *edge_pair_list* **)**

If a *period_multiplier* is specified, the period of the derived waveform is obtained by multiplying the period of the parent waveform by the value given in the *period_multiplier* construct. The position of each waveform edge in the parent is also multiplied, to determine the corresponding edge position in the derived waveform.

If a *phase_shift* is specified, the edges of the derived waveform are computed by adding the specified value to the edge positions specified in the parent waveform or to the computed edge positions if a *period_multiplier* is specified.

The values specified in the *skew_adjustment* construct are used to change the uncertainty region defined for each corresponding edge in the parent waveform, or for the computed edge positions if either a *period_multiplier* or a *phase_shift* is specified. If a single skew adjustment number is specified for an edge, it is subtracted from the left edge of the uncertainty region associated with the corresponding edge in the parent and added to the right edge of that uncertainty region. If two skew adjustment numbers are specified for an edge, the first number is subtracted from the left edge of the uncertainty region associated with the corresponding edge in the parent, and the second number is added to the right edge of that uncertainty region.

When a combination of *period_multiplier*, *phase_shift*, or *skew_adjustment* constructs are specified, the edge positions are computed by first considering the effect of any *period_multiplier*, then the effect of any *phase_shift*, and finally, the effect of any *skew_adjustment*.

The waveform resulting from the calculations must be valid: offsets must increase monotonically throughout the *edge_pair_list* and must not exceed the adjusted period.

When the **MULTI_CYCLE** construct (see “Multi-Cycle Paths” on page 83) is used for a parent waveform, it has no effect on any waveforms derived from that parent; any adjustments must be specified independently for each derived waveform.

Example

```
(LEVEL 1
  (DERIVED_WAVEFORM “50 MHz 50/50” “100 MHz 50/50”
    (period_multiplier 2)
  )
)
```

In this example, a waveform is defined with a 50% duty cycle and a 20 ns period by deriving from a previously defined parent waveform.

Clock Groups

By default, all clocks are assumed to be derived from a common source clock and to have harmonically related frequencies, so that it is meaningful to perform timing checks on paths between any pair of registers.

In Level 1, not all of the clocks need to be derived from the same source. In this case, the waveforms can be separated into groups of related clocks or “clock domains.” Only paths between clock waveforms in the same

group are constrained. In Level 0, all clock waveforms are assigned to the same default clock domain.

Clock waveforms in different domains are assumed to be asynchronous. There is no default constraint on the delay of paths which start in one clock domain and end in a different one, although an explicit combinational delay constraint could be specified as an exception. A synchronizer must usually be used for these paths.

Syntax

clock_group ::= **(** *label?* **CLOCK_GROUP** *clock_group_name waveform_name+* **)**

clock_group_name ::= QSTRING

The clocks within the group are identified by their waveform names, and the definitions of the waveforms must precede the *clock_group_spec*. Usually derived waveforms will be in the same clock group as their parent waveform, but this must be specified explicitly.

Including the same waveform name in multiple clock groups is not allowed because doing so implies that the clock is asynchronous with respect to itself.

Example

```
(WAVEFORM “100 MHz 50/50” 10.0
  (posedge 0) (negedge 5.0)
)

(LEVEL 1
  (DERIVED_WAVEFORM “50 MHz 50/50” “100 MHz 50/50”
    (period_multiplier 2)
  )
  (CLOCK_GROUP “group1”
    “100 MHz 50/50” “50 MHz 50/50”
  )
)
```

Timing Globals Case

The timing globals can be case-dependent.

Syntax

timing_globals_case ::= **(** **CASE** IDENTIFIER *timing_globals_case_spec+* **)**

timing_globals_case_spec ::= *timing_globals_spec_0*
||= *timing_globals_no_case_1*

Example

```
(GLOBALS_SUBSET TIMING
  (level 1
    (case board
      (WAVEFORM “100 MHz 50/50” 10.0
        (posedge 0) (negedge 5.0)
      )
    )
    (case tester
      (WAVEFORM “20 MHz 50/50” 10.0
        (posedge 0) (negedge 5.0)
      )
    )
  )
)
```

In this example, the clock waveform supplied to the chip depends on whether it is mounted on the board or is being tested.

Design References

GCF allows three types of design preferences: name prefixes, cell and port instances, and cell types.

Name Prefix

Constraints generally refer to the properties of specific objects within a design (for example, cell instances or port instances). In GCF, it is only possible to refer to these objects by their name. However, the full hierarchical name of a design object can be a fairly long string, and many design objects have similar names.

To reduce the size of GCF files, a notation is adopted which is similar to one originally defined for the Physical Design Exchange Format, Revision 2.0 (PDEF).

To reduce the size of GCF files, GCF allows the use of name prefixes. A name prefix is a short alias to be created for an initial portion of a hierarchical path name. When the full hierarchical names of many design objects share a common initial prefix, the use of name prefixes can substantially reduce the size of a GCF file.

Syntax

name_prefixes ::= **(NAME_PREFIXES** *num_prefixes* *name_prefix+* **)**

num_prefixes ::= DNUMBER

name_prefix ::= *prefix_id* QSTRING

prefix_id ::= DNUMBER

To optimize reading a GCF file, the *num_prefixes* parameter must specify the exact number of name prefixes which follow, and the *prefix_ids* must be consecutive integers starting at 0.

Name prefixes are defined within a cell specification. A GCF writer can choose to use any set of strings for use as name prefixes, or can choose to not define any prefixes at all. One possible choice for the name prefixes is the instance names of primitives instantiated as descendents of the cell.

Once a name prefix has been defined, it can be used to identify cell instances or port instances within the current cell instance. The definition of the name prefix must precede any usage of the prefix.

When a name prefix is used, it is interpreted as the initial portion of a relative path name beginning at the context of the current cell instance.

Cell and Port Instance

The cell instance construct is used to identify a particular instance of a cell within the design. The port instance construct is used to identify a particular instance of a port within the design.

Syntax

cell_instance ::= PATH
||= **(** *prefix_id* **)**
||= **(** *prefix_id* PARTIAL_PATH **)**

port_instance ::= *port*
||= PATH HCHAR *port*
||= **(** *prefix_id port* **)**
||= **(** *prefix_id* PARTIAL_PATH HCHAR *port* **)**

Since the name prefix and the PARTIAL_PATH are simply concatenated without interpretation to form the full PATH for the cell or port instance, the name prefix must use the hierarchy delimiter character, HCHAR, to separate each level of hierarchy in the name.

There must be no white space separating the PATH or PARTIAL_PATH, HCHAR, and *port* components of a *port_instance*.

Example

```
(CELL()
  (NAME_PREFIXES 2
    0 “a.b.c.d.”
    1 “a.b.c.e.”
  )
  (SUBSET “timing”
    (MAX_TRANSITION_TIME 1.0 2.0 (1 IN1))
    (MAX_TRANSITION_TIME 3.0 4.0 (2 IN1))
  )
)
```

In this example, two name prefixes are defined and then used to construct the full path name for two different input port instances to set a transition time constraint on those ports.

Cell Type

The *cell_id* construct is used to refer to exactly one type of cell.

Syntax

cell_id ::= **(CELLTYPE** *cell_name* **)**
||= **(CELLTYPE**
library_name cell_name view_name? **)**

library_name ::= QSTRING

cell_name ::= QSTRING

view_name ::= QSTRING

The library name indicates the library which contains the cell. The view name specifies a particular view of the cell.

Cell Entries

A cell construct identifies a particular “region” or “scope” within a design and contains constraint data to be applied to that region.

For example, a cell construct might identify a unique occurrence of a user-defined cell or block and provide constraints on the interface ports of that block. Or, it might identify a unique occurrence of an ASIC physical primitive (such as a flip-flop) in the design and define constraints specific to that occurrence (such as a multi-cycle path constraint on all paths starting at that flip-flop). Besides identifying such design-specific regions, cell entries can identify all occurrences of a particular user-defined cell or an ASIC library physical primitive, such as a certain type of gate or flip-flop. Data is applied to all such regions in the design.

Syntax

cell_spec ::= **(CELL** *cell_instance_spec cell_body_spec+* **)**

cell_instance_spec ::= *cell_instance_path*
||= **(** *cell_instance_path+* **)**
||= **( )**
||= *cell_views*

cell_instance_path ::= PATH

cell_body_spec ::= *name_prefixes*
||= *subset*
||= *extension*
||= *meta_data*
||= *include*

The *cell_instance_spec* identifies one or more regions of the design. The *cell_body_spec* contains the constraint data for that region. These will be discussed in detail in the following chapters.

Example

```
(CELL a1.b1.c1
  (SUBSET PARASITICS
    (INTERNAL_LOAD 5.0 7.5 IN1)
  )
)
```

A GCF file can contain any number of cell entries (including zero). The order of the cell entries is significant only if they have an overlapping effect, where data from two different cell entries applies to the same constraint in the design. In this situation, the cell entries are processed strictly from the beginning to the end of the file, and the data they contain

is applied in sequence to whatever region is appropriate to that cell construct. Where data is applied to a constraint previously referenced by the same GCF file, the new data will be applied over the old.

This interpretation supports the definition of a set of default constraints for all instances of a cell, then overriding those constraints for particular cell instances.

Cell Instance Spec

The *cell_instance_spec* identifies the parts of the design to which the constraints in the cell construct apply.

Syntax

cell_instance_spec ::= *cell_instance_path*
||= **(** *cell_instance_path*+ **)**
||= **( )**
||= *cell_views*

cell_instance_path ::= PATH

The first form of the *cell_instance_spec* identifies a unique occurrence in the design. The *cell_instance_path* must be relative to the level in the design at which the annotator is instructed to apply the GCF file (see “The Annotator” on page 21). Frequently, this is the topmost level.

The *cell_instance_path* is extended down through the hierarchy by specifying a hierarchical path name with the name of each hierarchical level separated by the hierarchy delimiter character, HCHAR. The hierarchical path name must not start with the hierarchy delimiter character. Name prefixes cannot be used in the *cell_instance_path.*

Example

```
(CELL a1.b1.c1
   . . .
)
```

In this example, the relative hierarchical path is specified as `a1.b1.c1`. The region identified is cell or block `c1` within block `b1`, which is in turn within block `a1`, which must be contained within the level at which the GCF is applied. The period character separates levels or elements of the path. The example assumes that the delimiters construct in the GCF header specified the hierarchy delimiter as the period character or, since period is the default, the construct was absent.

The second form of the *cell_instance_spec* identifies several occurrences of the cell to which the same constraints must be applied.

The () form of the *cell_instance_spec* indicates that the constraints defined in the *cell_body_spec* apply to the hierarchical level in the design at which

the annotator is instructed to apply the GCF file. This is typically used to specify constraints on the top-level cell in the design.

The *cell_views* form of the cell instance list indicates that the constraints defined within the *cell_body_spec* apply to all occurrences of the given type of cell which are instantiated under the hierarchical level at which the GCF is applied.

Syntax

cell_views ::= **(CELLTYPE** *cell_name* **)**
||= **(CELLTYPE**
*library_name cell_name view_name** **)**

library_name ::= QSTRING

cell_name ::= QSTRING

view_name ::= QSTRING

The library name indicates the library which contains the cell, while the view name can be used to specify which views of the cell are affected.

Example

```
(CELL (CELLTYPE “WORKLIB” "ALU")
   . . .
)
```

The effect of this example is to apply the constraints to every instance of every view of the ALU cell from the WORKLIB library.

Subsets

GCF is organized into a number of subsets of related constraint data. The intent of this is to allow the development of the GCF standard for each subset to proceed independently, and to allow tools to efficiently access only the data which is relevant to them.

Syntax

subset ::= *timing_subset*
||= *parasitics_subset*
||= *area_subset*
||= *power_subset*

Additional subsets are likely to be added in the future.

4

Timing Subset

Timing Subset Header

Timing Environment

Timing Exceptions

Timing Subset Header

The timing subset of each cell entry in the GCF file includes information about the following:

- The timing environment in which the cell is intended to operate
- The constraints on the timing characteristics of the cell

This chapter describes the timing environment and timing exceptions. For information on other constructs, refer to “Extensions” on page 37, “Meta Data” on page 40, and “Include Files” on page 42.

Syntax

timing_subset ::= **(SUBSET TIMING** *timing_subset_body* **)**

timing_subset_body ::= *timing_subset_spec+*
||= *include*

timing_subset_spec ::= *timing_environment*
||= *timing_exceptions*
||= *extension*
||= *meta_data*

Example

```
(CELL
  (CELLTYPE "WORKLIB" "ALU")
  (INSTANCE *)
  (SUBSET TIMING
    (ENVIRONMENT
       . . .
    )
    (EXCEPTIONS
       . . .
    )
  )
)
```

Timing Environment

The timing environment of a cell describes a number of conditions external to the cell that affect its timing behavior. The following conditions are included:

- Arrival and departure times of signals at the cell ports
- Clock waveforms used by the cell
- Information about the external drivers connected to the input ports of the cell

This section describes clock specifications, arrival time, driver cell, driver strength, input slew, constant values, operating conditions, and timing environment cases. Chapter 5, “Parasitics Subset,” includes additional information that affects the cell’s timing behavior.

Syntax

timing_environment ::= **(ENVIRONMENT** *timing_env_spec+* **)**

timing_env_spec ::= *timing_env_spec_0*
||= *timing_env_spec_1*

timing_env_spec_0 ::= *clock_spec*
||= *arrival_spec*
||= *departure_spec*
||= *external_delay_spec*
||= *driver_spec*
||= *input_slew_spec*
||= *extension*
||= *meta_data*

timing_env_spec_1 ::= **(LEVEL 1** *timing_env_1+* **)**

timing_env_1 ::= *timing_env_no_case_1*
||= *timing_env_case*

timing_env_no_case_1 ::= *constant_spec*
||= *operating_conditions*
||= *internal_slew_spec*

Clock Specifications

Each clock that is applied to the cell (or generated internally by the cell itself) is described by relating a waveform (see “Timing Globals” on page 50) to a port instance (the source of that waveform within the cell). These port instances are usually the roots of a clock network and are referred to as clock roots.

Syntax

clock_spec ::= **(** *label?* **CLOCK** *waveform_name*
port_instance+ **)**

If the waveform was not previously defined, an error message will be given. Although the **WAVEFORM** construct generally allows more than one pair of edges, clock waveforms must only have a single pair of edges.

The *port_instance* can either be an input port on the cell or an output port of an instance within the cell.

Example

```
(CLOCK “100 MHz 50/50” clk1)
(CLOCK “50 MHz 50/50” divider.clkout)
```

Arrival Time

The **ARRIVAL** construct defines ranges of time in which signal transitions can occur at a *port_instance* which includes registers in its transitive fanout. Arrival times are usually specified only for primary input and bidirectional ports, but they can also be specified for internal input and bidirectional ports. When specified on internal pins, the arrival time overrides any propagated arrival time.

Syntax

arrival_spec ::= **(** *label?* **ARRIVAL**
waveform_edge arrival_value
*port_instance** **)**

waveform_edge ::= **(** *waveform_edge_identifier waveform_name* **)**

arrival_value ::= **(** *waveform_edge_identifier r_min_max* **)**
||= *r_rise_fall_min_max*

If no *port_instance* is specified, the arrival time applies by default to all primary input and bidirectional ports on the cell except those which have been identified as clock inputs.

The *waveform_edge* specification, which identifies a waveform and an edge of that waveform, is required. The *arrival_value* is added to that edge.

If the waveform was not previously defined, an error message will be given. Although the **WAVEFORM** construct generally allows more than one pair of edges, clock waveforms used for arrival times must only have a single pair of edges.

The first *arrival_value* form, *waveform_edge_identifier r_min_max*, must be used to specify the arrival time of just the rising edges or just the falling edges. The second form, *r_rise_fall_min_max*, must be used to specify the arrival time of both rising and falling edges.

One or two values can be specified for the *r_min_max* form. If a single value is specified, it applies to both the minimum and maximum values. If

two values are specified, they represent the minimum and maximum values, respectively.

One, two, or four values can be specified for the *r_rise_fall_min_max* form. If a single value is specified, it applies to the rise minimum, rise maximum, fall minimum, and fall maximum values. If two values are specified, the first value applies to the rise minimum and rise maximum values, and the second value applies to the fall minimum and fall maximum values. If four values are specified, they apply to the rise minimum, rise maximum, fall minimum, and fall maximum values, respectively.

The minimum values must be less than or equal to the maximum values for the same transition.

Multiple **ARRIVAL** constructs can be defined for the same port. Each **ARRIVAL** construct can reference a different *waveform_edge*. The arrival times associated with a given reference *waveform_edge* are independent of the arrival times associated with any other reference *waveform_edge*, and analysis will be done separately for each reference *waveform_edge*.

If several **ARRIVAL** constructs appear in a GCF file, and each construct specifies arrival times for the same port instance with respect to the same reference *waveform_edge*, the effect is cumulative and overriding. For example, assume there are two arrival constructs for the same port instance with respect to the same reference *waveform_edge*:

- If the first construct specifies only the **POSEDGE** arrival times and the second construct specifies only the **NEGEDGE** arrival times, the result is that both the **POSEDGE** and **NEGEDGE** arrival times are set.
- If the first construct specifies both **POSEDGE** and **NEGEDGE** arrival times and the second construct specifies only the **NEGEDGE** arrival times, the result is that the values of the **POSEDGE** arrival times come from the first construct, while the values of the **NEGEDGE** arrival times come from the second construct.

Example

```
(ENVIRONMENT
  (ARRIVAL (POSEDGE “50 MHz 50/50”)
    10.0 14.0 12.0 16.0  D[*] )
)
```

This example specifies the arrival times for all input pins referenced by the bit-spec `D[*]`. Assuming that the time scale is in ns, rise transitions will occur no sooner than 10 ns and no later than 14 ns after the rising edge of

the reference clock. Falling transitions will occur no sooner than 12 ns and no later than 16 ns after the clock edge.

Departure Time

The **DEPARTURE** construct defines ranges of time in which signal transitions must occur at a *port_instance* which includes registers in its transitive fanin. Departure times are usually specified only for primary output and bidirectional ports, but they can also be specified for internal output and bidirectional ports. When specified on internal pins, the departure time overrides any propagated departure time.

Syntax

departure_spec ::= **(** *label?* **DEPARTURE**
waveform_edge departure_value
*port_instance** **)**

waveform_edge ::= **(** *waveform_edge_identifier waveform_name* **)**

departure_value ::= *setup_rise_fall hold_rise_fall*
||= **(** *waveform_edge_identifier*
setup_value hold_value **)**

setup_rise_fall ::= *r_rise_and_fall*

hold_rise_fall ::= *r_rise_and_fall*

setup_value ::= RNUMBER

hold_value ::= RNUMBER

If no *port_instance* is specified, the departure time applies by default to all primary output and bidirectional ports on the cell.

The *waveform_edge* specification, which identifies a waveform and an edge of that waveform, is required. The hold *departure_value* is added to that edge, while the setup *departure_value* is subtracted from that edge.

If the waveform was not previously defined, an error message will be given. Although the **WAVEFORM** construct generally allows more than one pair of edges, clock waveforms used for departure times must only have a single pair of edges.

Departure times are interpreted as setup and hold constraints. Specifying a departure time is equivalent to adding a register with corresponding setup and hold constraints at the output.

All partial paths from the specified port to the target registers must be considered in setting the departure time.

- For the minimum departure time, the delay of each partial path must be subtracted from the hold time of the target register, and the minimum departure time must be set to the largest (most positive) resulting value.

Since the partial path delays will generally be larger than the hold time of the target registers, the minimum departure time will usually be a negative number.

- For the maximum departure time, the setup time of the target register must be added to the delay of each partial path, and the maximum departure time must be set to the largest resulting value.

The first *departure_value* form, *setup_rise_fall hold_rise_fall*, must be used to specify different departure times for rising and falling edges. The second form, *waveform_edge_identifier setup_value hold_value*, must be used to specify the departure time of just the rising edges or just the falling edges.

Multiple **DEPARTURE** constructs can be defined for the same port. Each **DEPARTURE** construct can reference a different *waveform_edge*. The departure times associated with a given reference *waveform_edge* are independent of the departure times associated with any other reference *waveform_edge*, and analysis will be done separately for each reference *waveform_edge*.

Like **ARRIVAL** constructs, the effect of multiple **DEPARTURE** constructs is cumulative and overriding.

Example

```
(ENVIRONMENT
  (DEPARTURE (NEGEDGE “50 MHz 50/50”)
    12.0 18.0 -8.0 -14.0 A[15:0] )
)
```

This example specifies departure times for each of the 16 output pins `A[15:0]` and that the falling edge is the active edge of the target clock. Assuming that the time scale is in ns, rising transitions must occur no later than 12.0 ns before the setup active edge and no earlier than 8.0 ns before the hold active edge. Falling transitions must occur no later than 18.0 ns before the setup active edge and no earlier than 14.0 ns before the hold active edge.

External Delay

The **EXTERNAL_DELAY** construct is used with the **PATH_DELAY** construct to constrain purely combinational portions of a design.

The **PATH_DELAY** construct describes constraints on the combinational delay through a portion of the design, while the **EXTERNAL_DELAY** construct describes purely combinational delays which are external to that portion of the design. The external delays are added to the computed path delays within that portion of the design before comparing to the path delay constraint.

External delays may be specified on both primary interface ports and on internal ports. If no external delay is specified for a port which is an endpoint of a **PATH_DELAY** constraint, the external delay defaults to 0.

Syntax

external_delay_spec ::= **(** *label?* **EXTERNAL_DELAY** *external_delay_value endpoints_spec+* **)**

external_delay_value ::= **(** *waveform_edge_identifier r_min_max* **)**
||= *r_rise_fall_min_max*

The *endpoints_spec* is described in “Path Specifications” on page 78. External delays specified using the **FROM** keyword are to be added to combinational paths which start at the given endpoints, while external delays specified using the **TO** keyword are to be added to combinational paths which end at the given endpoints. A given internal port instance or primary bidirectional port can appear in two different external delay specifications, one using the **FROM** keyword and one using the **TO** keyword.

The first *external_delay_value* form, *waveform_edge_identifier r_min_max*, must be used to specify the external delay for just the rising edges or just the falling edges. The second form, *r_rise_fall_min_max*, must be used to specify the arrival time of both rising and falling edges. The transitions are with respect to the given endpoints.

One or two values can be specified for the *r_min_max* form. If a single value is specified, it applies to both the minimum and maximum values. If two values are specified, they represent the minimum and maximum values, respectively.

One, two, or four values can be specified for the *r_rise_fall_min_max* form. If a single value is specified, it applies to the rise minimum, rise maximum, fall minimum, and fall maximum values. If two values are specified, the first value applies to the rise minimum and rise maximum values, and the second value applies to the fall minimum and fall maximum values. If four values are specified, they apply to the rise minimum, rise maximum, fall minimum, and fall maximum values, respectively.

The minimum values must be less than or equal to the maximum values for the same transition.

Like **ARRIVAL** and **DEPARTURE** constructs, the effect of multiple **EXTERNAL_DELAY** constructs for the same port instance is cumulative

and overriding.

Example

```
(ENVIRONMENT
  (EXTERNAL_DELAY 5.0
    (FROM IN[0] )
  )
  (EXTERNAL_DELAY 3.0
    (TO OUT[0] )
  )
  (PATH_DELAY 10.0
    (FROM IN[0] )
    (TO OUT[0] )
  )
)
```

Assuming that time values are in ns, this example specifies that

- An external combinational delay of 5 ns should be added to the computed delay of any purely combinational path starting at `IN[0]`
- An external combinational delay of 3 ns should be added to the computed delay of any purely combinational path ending at `OUT[0]`
- The effective combinational delay constraint for paths starting at `IN[0]` and ending at `OUT[0]` is 2 ns (the 10 ns **PATH_DELAY** constraint minus the two external delays).

Driver Specification

Driver specifications describe information about an external driver which is connected to a primary input or bidirectional port of the cell.

Syntax

driver_spec ::= *driver_cell_spec*
||= *driver_strength_spec*

Precedence Rules

There are several different types of driver specifications, as well as the ability to directly specify the slew for an input. When several different constructs appear in a GCF which affect a given port, the following rules are used to determine which of the constructs should be used:

- An explicit specification of the driver cell, driver strength, or implicit slew for a given port always overrides any of the defaults.
- When there are multiple explicit specifications for the same port, the precedence (in decreasing order) is driver cell, input slew, driver strength.

- When there are multiple default specifications, but no explicit specifications for a given port, the precedence (in decreasing order) of the defaults is also driver cell, input slew, driver strength.

Driver Cell

The **DRIVER_CELL** construct is used when the cell type of the external driver is known. For example, for a user-defined block within a chip, the external driver is usually a cell within another user-defined block. The default driver cell type can be specified for all primary input and bidirectional ports by not specifying any *port_instance*.

Syntax

driver_cell_spec ::= **(** *label?* **DRIVER_CELL**
driver_cell_port_spec
driver_cell_options?
opt_port_instance_list **)**

driver_cell_port_spec ::= **(** *cell_id* **)**
||= **(** *cell_id output_port* **)**
||= **(** *cell_id input_port output_port* **)**

driver_cell_options ::= **(** *driver_cell_option+* **)**

driver_cell_option ::= *drive_multiplier*
||= *driver_input_slew*
||= *waveform_edge_identifier*

drive_multiplier ::= **(** **PARALLEL_DRIVERS** DNUMBER **)**

driver_input_slew ::= **(** **INPUT_SLEW** *slew_value input_port** **)**

slew_value ::= *rise_fall_min_max*

If a *waveform_edge_identifier* is specified, the driver cell construct only applies to delay calculation for that edge.

If multiple buffers of the same type are connected in parallel, the number of those buffers can be specified using the **PARALLEL_DRIVERS** construct. If multiple buffers of different types are connected in parallel, multiple **DRIVER_CELL** constructs can be specified. When a driver cell type is explicitly specified for a primary input and bidirectional port, it overrides any default; the explicitly specified driver cell is not connected in parallel with the default driver cell.

The *output_port* specifies the port on the driving cell that is connected to the primary inputs. It must be specified whenever the driving cell has multiple outputs.

The *input_port* specifies a single input port on the driving cell that must be the starting point when doing delay calculation. If the *input_port* is not specified, delay calculation is done by computing the worst case across all inputs ports that are associated with the specified *output_port*.

Input slews can be specified for one or more of the input ports on the driver. If the input slew is not specified for an input port that is the starting point for a timing arc considered in delay calculation, a default slew of 0 is used.

The *slew_values* are time values and must be specified in the units defined by the *time_scale*. The voltage thresholds for measuring the slew are defined by the **VOLTAGE_THRESHOLD** construct (see "Voltage Threshold" on page 48). If no voltage thresholds are specified, the *slew_value* represents by default the time required to transition between the 10 and 90 percent points of the power supply voltage.

One, two, or four values can be specified for the *slew_value*. If a single value is specified, it applies to the rise minimum, rise maximum, fall minimum, and fall maximum values. If two values are specified, the first value applies to the rise minimum and rise maximum values, and the second value applies to the fall minimum and fall maximum values. If four values are specified, they apply to the rise minimum, rise maximum, fall minimum, and fall maximum values, respectively.

The information about the driver cell affects the accuracy of the delay calculation.

- For the most accurate approach, both the *input_port* and the *output_port* must be provided, along with the slew at the *input_port*. In general, this is only feasible when there is only one connected input port. At the time a GCF file is created, it is unknown which input port is switching, and a worst-case analysis must be done instead.
- For the most accurate worst-case analysis, the *output_port* on the driver cell must be specified, along with the slew at every input.
- For a less accurate worst-case analysis, the slew values for each input port can be omitted, in which case the default slew is used.

Driver Strength

When the cell type of the external driver is not known, the **DRIVER_STRENGTH** construct can be used instead.

Syntax

driver_strength_spec ::= **(** *label?* **DRIVER_STRENGTH** *strength_value* *port_instance** **)**

strength_value ::= *rise_fall*

The default driver strength can be specified for all primary input and bidirectional pins by not specifying any *port_instance*.

The *strength_values* are resistance values and must be specified in the units defined by the *res_scale*. One or two values can be specified for the *strength_value*. If a single value is specified, it applies to both rise and fall. If two values are specified, they apply to rise and fall, respectively.

Input Slew

When the cell type of the external driver is not known, the **INPUT_SLEW** construct can be used instead. Note that the **INPUT_SLEW** construct can be used both within the context of a **DRIVER_CELL** construct and by itself. When used by itself, it describes the input slew at the primary input of the cell, and a label can be associated with the construct.

Syntax

input_slew_spec ::= **(** *label?* **INPUT_SLEW** *slew_value* *port_instance** **)**

The default input slew can be specified for all primary input and bidirectional pins by omitting the *port_instances*.

The *slew_value* is a time value and must be specified in the units defined by the *time_scale*. The voltage thresholds for measuring the slew are defined by the **VOLTAGE_THRESHOLD** construct (see “Voltage Threshold” on page 48). If no voltage thresholds are specified, the *slew_value* represents by default the time required to transition between the 10 and 90 percent points of the power supply voltage.

Constant Values

In Level 1, GCF allows specifying that certain signals have a constant value. Often, this is used to describe case-dependent constraints (see “Cases” on page 35) or to disable a portion of a circuit.

Syntax

constant_spec ::= **(** **CONSTANT** *constant_value port_instance+* **)**

constant_value ::= **0**
||= **1**

Constant values are defined in terms of signals but specified using *port_instances*. A constant value specified for any of the *port_instances* connected to a signal affects the signal as a whole. An error message will be given if different constant values are specified on two *port_instances* connected to the same signal.

Operating Conditions

The operating conditions defined in the global environment subset (see “Environment Globals” on page 45) apply by default to all cells in the design. These conditions can be overridden for particular cells by including an *operating_conditions* specification in the timing subset for a cell. When applied to a non-leaf cell, the operating conditions are

overridden for that cell and all of its descendents, unless overridden again by one of the descendents.

Internal Slew

The **INTERNAL_SLEW** construct is a Level 1 construct and specifies a slew that overrides the default slew on internal pins (input or bidirectional pins on primitives). Normally, **INTERNAL_SLEW** must not be used for clock input pins on primitives; the **SLEW** option of the **CLOCK_DELAY** construct must be used instead.

Syntax

internal_slew_spec ::= **(** *label?* **INTERNAL_SLEW** *rise_fall* *port_instance** **)**

The **INTERNAL_SLEW** construct is normally only used

- For input or bidirectional pins that are part of a combinational loop broken using a disable
- For cases where the slew that would be computed by the normal delay calculation is known to be inaccurate

The default internal slew can be set by not specifying any *port_instance*.

The internal slew values will be determined using the following precedence order:

- An explicit **INTERNAL_SLEW** for the pin
- The calculated slew, if it is possible to calculate one
- The default **INTERNAL_SLEW**, if no slew can be calculated
- The default **INPUT_SLEW**
- 0

Timing Environment Cases

The timing environment can be case-dependent.

Syntax

timing_env_case ::= **(** **CASE** IDENTIFIER *timing_env_case_spec+* **)**

timing_env_case_spec ::= *timing_env_spec_0*
||= *timing_env_no_case_1*

Example

```
(ENVIRONMENT
  (level 1
    (case board
      (input_slew 2.0 1.0 in1)
    )
    (case tester
      (input_slew 5.0 3.0 in1)
    )
  )
)
```

In this example, the input slew of a signal supplied to the chip depends on whether the chip is mounted on the board or is being tested.

Timing Exceptions

By default, GCF assumes that, a circuit is synchronous. This assumption implies that there are a set of implicit constraints on the delays of paths through combinational logic. These constraints are determined by the clock waveforms provided to source registers and target registers, and by the arrival and departure times specified for ports on the cell.

Timing exceptions are GCF constructs that can be used to

- Override the implicit synchronous timing constraints for portions of a design
- Describe explicit constraints on asynchronous portions of a design

This section describes path specifications, disable specifications, multi-cycle paths, combinational delays, max transition times, internal slew, latch-based borrowing, clock delay, and timing exception cases.

Syntax

timing_exceptions ::= **(EXCEPTIONS** *timing_exception_spec+* **)**

timing_exception_spec ::= *timing_exception_spec_0*
||= *timing_exception_spec_1*

timing_exception_spec_0 ::= *disable_spec_0*
||= *multi_cycle_spec_0*
||= *path_delay_spec_0*
||= *transition_time_spec*
||= *extension*
||= *meta_data*

timing_exception_spec_1 ::= **(LEVEL 1** *timing_exception_1+* **)**

timing_exception_1 ::= *timing_exception_no_case_1*
||= *timing_exception_case*

timing_exception_no_case_1 ::= *disable_spec_1*
||= *multi_cycle_spec_1*
||= *path_delay_spec_1*
||= *borrow_limit_spec*
||= *clock_delay_spec*

Path Specifications

Many of the timing exceptions require path specifications. This section describes the various ways of specifying paths.

Syntax

```
arc_spec       ::= ( ARC port_instance port_instance )
thru_spec      ::= ( THRU  port_instance )
thru_all_spec  ::= ( THRU_ALL port_instance port_instance+ )
endpoints_spec ::= from_spec
               ||= to_spec
               ||= ( from_spec to_spec )
from_spec      ::= ( FROM from_to_item+ )
to_spec        ::= ( TO from_to_item+ )
from_to_item   ::= port_instance
               ||= cell_instance
               ||= waveform_name
```

The Level 0 **ARC** construct specifies all paths that pass through both of the *port_instances*, including paths which start or end at the arc. The port instances must be contiguous in the path (either an input to output connection on a cell, or an output to input connection on a net). The SDF **IOPATH** and **INTERCONNECT** constructs describe similar arcs.

The Level 0 **THRU** construct specifies all paths that pass through the given port, including those which start or end at the port.

The Level 0 **THRU_ALL** construct specifies all paths that pass through all of the ports listed. These ports do not have to be contiguous in the paths, but they must be listed in the order in which they would be encountered in traversing each path from the source to the target.

The *endpoints_spec* specifies all paths that start at any of the **FROM** items and end at any of the **TO** items. The **FROM** items must be waveform names, primary input or bidirectional ports, registers, register clock inputs, or register data outputs. The **TO** items must be waveform names, primary output or bidirectional ports, registers, register clock inputs, or register data inputs.

Disable Specifications

Disabling paths is important for the following reasons:

- To break combinational feedback loops
- To eliminate false paths (paths that will never be activated during normal operation of the circuit)
- To eliminate paths that are only active during certain modes of circuit operation (for example, paths associated with testability logic)

The **DISABLE** construct identifies a set of paths for which selected timing checks must be suppressed.

The timing checks that might be affected are separated into two groups:

- The minimum timing checks are hold, removal, and the hold portion of no-change checks. When the **HOLD** keyword is specified in a disable construct, it refers generically to all of the minimum timing checks.
- The maximum timing checks are setup, recovery, and the setup portion of no-change checks. When the **SETUP** keyword is specified in a disable construct, it refers generically to all of the maximum timing checks.

In the context of disabled paths, the phrase “all timing checks” means both minimum and maximum timing checks, but not skew, period, or pulse width checks.

Level 0 Disables

In Level 0, the paths can be identified by a cell instance, a single port instance, an arc, or the path endpoints.

Syntax

disable_spec_0 ::= *disable_item_spec_0*
||= *disable_endpoints_spec_0*

Disabling Paths Identified by Items

The simplest form of the **DISABLE** construct, *disable_item_spec_0,* disables all timing checks associated with a set of paths.

Syntax

disable_item_spec_0 ::= **(** *label?* **DISABLE** *disable_item_0+* **)**

disable_item_0 ::= *port_instance*
||= *cell_instance*
||= *arc_spec*

If a *port_instance* is specified, all timing checks associated with paths through that port instance are disabled. The following types of *port_instances* are handled differently:

- If the *port_instance* is a clock input, all timing checks related to that clock input or associated with paths that begin at that clock input are disabled.
- If the *port_instance* is an enable or disable pin that affects other paths through a cell instance (such as a latch enable or a tri-state enable), the timing checks associated with paths through that pin are affected, but not the timing checks associated with paths which it controls.

If a *cell_instance* is specified, all output ports on that cell instance are implicitly referenced. All timing checks associated with any paths which start at or pass through any output port on the cell instance are disabled.

The timing checks on paths that end at input ports on the cell instance are not affected (paths through input ports to output ports are affected).

If a Level 0 **ARC** construct is given, all timing checks associated with any paths that pass through the arc are disabled, including paths that either start or end at the arc. For more information, refer to “Path Specifications” on page 78.

Disabling Paths Identified by Endpoints

The *disable_endpoints_spec_0* construct disables selected timing checks on a set of paths that are identified by their from, to, or from and to endpoints.

Syntax

disable_endpoints_spec_0 ::= **(** *label?* **DISABLE** *endpoints_spec+*
disable_endpoints_options? **)**

disable_endpoints_options ::= *timing_check*
||= *edge_identifier*
||= *timing_check edge_identifier*

timing_check ::= **SETUP**
||= **HOLD**

If the **SETUP** or **HOLD** keyword is specified, only the maximum or the minimum timing checks must be disabled; otherwise, both the maximum and minimum timing checks are disabled.

If an *edge_identifier* is specified, the selected timing checks are disabled only for the specified edge of the signal, as measured at the path endpoint.

Level 1 Disables

In Level 1, the timing checks or edges that are affected by a port instance or arc disable can be selected. The paths can be identified by multiple ports, cell instance name, or cell type.

Syntax

disable_spec_1 ::= *disable_edges_spec_1*
||= *disable_cell_spec_1*

Disabling Paths Associated With Port Instances

The *disable_edges_spec_1* construct disables selected timing checks on a set of paths. If the **SETUP** or **HOLD** keyword is specified, only the maximum or the minimum timing checks must be disabled; otherwise, both the maximum and minimum timing checks are disabled.

Syntax

```
disable_edges_spec_1 ::= ( label? DISABLE
                             disable_edges_path_spec+
                             timing_check? )
disable_edges_path_spec ::= thru_edge_spec
                        ||= arc_edges_spec
                        ||= thru_all_edges_spec
thru_edge_spec ::= ( THRU port_instance_edge )
arc_edges_spec ::= ( ARC port_instance_edge port_instance_edge )
thru_all_edges_spec ::= ( THRU_ALL
                             port_instance_edge port_instance_edge+ )
port_instance_edge ::= ( edge_identifier port_instance )
```

The Level 1 **THRU** construct specifies all paths which pass through a single port instance, including those which begin or end at the port instance. The selected timing checks are disabled only for the specified edge of the signal, as measured at that port instance and propagated to each target in the set of paths. The following types of *port_instances* are handled differently.

- If the *port_instance* is a clock input, all selected timing checks related to that clock input or associated with paths which begin at that clock input are disabled.
- If the *port_instance* is an enable or disable pin that affects other paths through a cell instance (such as a latch enable or a tri-state enable), the selected timing checks associated with paths through that pin are affected, but not the timing checks associated with the paths controlled by the pin.

The Level 1 **ARC** construct specifies all paths that pass through the given arc, including those which begin or end at the arc. The selected timing checks are disabled only for the specified edges of the signal, as measured at the start and end of the arc and propagated to each target in the set of paths.

The Level 1 **THRU_ALL** construct specifies all paths that pass through the specified port instances. The port instances do not have to be contiguous in the paths, but they must be listed in the order in which they would be encountered in traversing each path from the source to the target. The selected timing checks are disabled only for the specified edges of the signal, as measured at each *port_instance* and propagated to each target in the set of paths.

Disabling Paths Associated With Cell Instances or Cell Types

Syntax

disable_cell_spec_1 ::= **(** *label?* **DISABLE** *disable_cell_path_spec+* **)**

disable_cell_path_spec ::= *disable_instance_spec*
||= *disable_master_spec*

disable_instance_spec ::= **(INSTANCE** *cell_instance+* **)**

disable_master_spec ::= **(MASTER** *cell_id* **)**

The *disable_cell_spec_1* construct disables all timing checks associated with all paths associated with one or more cell instances including the following:

- All timing checks associated with paths to, from, or through the instance
- All timing checks associated with paths contained within the instance

Disabling a cell type affects all instances of that cell within either the current GCF cell instance or its descendents. All timing checks associated with all paths associated with any of those instances are disabled.

If a cell type is disabled within the GCF section for the top-level cell of a design, the cell type is disabled throughout the entire design.

Multi-Cycle Paths

The **MULTI_CYCLE** construct identifies the paths for which setup or hold checks must use a different set of active clock edges rather than the default. This construct is commonly used to describe paths whose data can propagate to the target register over multiple clock cycles by not clocking the target every cycle.

By default, timing checks are computed with respect to the active edges of the source and target clocks.For flip-flops, the active clock edge is the triggering clock edge. For level-sensitive latches, the active edges are the opening clock edge for sources and the closing clock edge for targets.

When the source and target clocks have the same frequency and phase, the following rules are commonly used to determine the active edges:

- Setup checks are computed between an active edge at the source in one cycle and the active edge at the target in the next cycle.
- Hold checks are computed between an active edge at the source in one cycle and the active edge at the target in the same cycle.

When the source and target clocks have different frequencies or phases, or when multiple cycles are allowed for a path, these rules can no longer be

used. A more precise definition of the process for choosing the default active edges is used in GCF.

Default Definition

The clock root that drives the source of a path is called the source clock root, and the waveform edge at the source clock root that triggers the source of a path is called the source root edge.

The clock root that drives the target of a path is called the target clock root, and the waveform edge at the target clock root that triggers the target of a path is called the target root edge.

If the clock signal is inverted between the clock root and the clock input of a register or latch, the root edge is different than the triggering edge of the register.

The relationship between particular source and target root edges determines which active edges are used for setup and hold checks. Multiple cycles of the source and target clocks are considered in identifying the source and target root edges for a timing check.

The setup check ensures that the expected data signals reach the target registers in time to be latched correctly. If no multi-cycle specification affects a path, the following rules are used for the setup check:

- Each target root edge and the nearest source root edge which precedes it are called a setup edge pair.
- The default source and target root edges are defined to be the setup edge pair with the smallest positive difference between the target root edge and the source root edge. The default active edges are the propagated versions of the root edges, measured at the source and target.

The hold check ensures that data does not reach the target registers early enough to be latched in the wrong cycle of the target clock. If no multi-cycle specification affects a path, every setup edge pair is considered for the hold check. For each setup edge pair, the root edges define the current cycle at the source and at the target. Two conditions must be satisfied with respect to these cycles:

- Data triggered by the current cycle at the source must not be latched by the previous cycle at the target. This condition defines a hold edge pair in which the hold source root edge is the same as the setup source root edge, and the hold target root edge is one cycle earlier than the setup target root edge.
- Data triggered by the next cycle at the source must not be latched by the current cycle at the target. This condition defines a hold edge pair

in which the hold source root edge is one cycle later than the setup source root edge, and the hold target root edge is the same as the setup target root edge.

These conditions are both checked by choosing the hold edge pair with the most positive difference between the target root edge and the source root edge (note that the difference can still be negative). The default active edges for the hold check are the propagated versions of the root edges, measured at the source and target.

Overriding the Default

The **MULTI_CYCLE** construct allows changing the active edges that are chosen for specific paths or for all paths between a given source and target clock pair.

Level 0 Multi-Cycle Paths

In Level 0, the paths can only be identified by their endpoints (see “Path Specifications” on page 78).

- The source endpoints (specified with the **FROM** construct) identify primary input or bidirectional ports or register clock inputs. In addition to explicitly identifying source endpoints, they can be specified implicitly using a register, register data outputs, or a waveform name.
  - If a register is specified, all clock inputs on the register are included as source endpoints.
  - If a register data output is specified, all clock inputs on the register that are related to that output are included as source endpoints.
  - If a waveform name is specified, all register clock inputs driven by the clock root(s) associated with the waveform are included as source endpoints, as are all primary input and bidirectional ports with an arrival time relative to that waveform.
- The target endpoints (specified with the **TO** construct) identify primary output or bidirectional ports or register data inputs. In addition to explicitly identifying target endpoints, they can be specified implicitly using a register, register clock inputs, or a waveform name.
  - If a register is specified, all data inputs on the register are included as target endpoints.
  - If a register clock input is specified, all data inputs on the register that are related to that clock input are included as target endpoints.
  - If a waveform name is specified, all data inputs on registers whose associated clock inputs are driven by the clock root(s) associated with the waveform are included as target endpoints, as are all

primary output and bidirectional ports with a departure time relative to that waveform.

- When both the source and target endpoints are specified using waveform names, the effect is to change the default relationship between the waveforms.

Syntax

multi_cycle_spec_0 ::= **(** *label?* **MULTI_CYCLE** *multi_cycle_option+ endpoints_spec+* **)**

multi_cycle_option ::= *timing_check_offset*
||= *edge_identifier*

timing_check_offset ::= **(** *timing_check num_cycles reference_clock?* **)**

reference_clock ::= **SOURCE**
||= **TARGET**

num_cycles ::= INUMBER

The *timing_check_offset*, which specifies the number of cycles to be allowed for a path, is used to adjust the active edges for the timing checks for all paths between the specified endpoints.

The following procedure is used to determine the setup edge pair:

- For all paths affected by a **MULTI_CYCLE** construct (whether **SETUP**, **HOLD**, or both **SETUP** and **HOLD** adjustments are specified), a default setup edge pair is chosen in the same way as for normal timing checks.
- Multiple cycles of the source and target clocks are still considered when determining the default setup edge pair. The pair with the smallest positive difference between the target root edge and the source root edge is selected.
- If the **SETUP** timing check is specified, then the corresponding *num_cycles* parameter is used to determine an adjusted setup edge pair as follows:
  - By default, or if **TARGET** is specified, the setup *num_cycles* parameter affects the target root edge. Instead of the default target root edge, the edge that arrives (*num_cycles* - 1) cycles later is used.
  - If **SOURCE** is specified, the setup *num_cycles* parameter affects the source root edge. Instead of the default source root edge, the edge that arrives (*num_cycles* - 1) cycles earlier is used.
- The adjusted active edges for the setup check are the propagated versions of the adjusted root edges, measured at the source and target.

The default hold edge pair is chosen differently for paths affected by a **MULTI_CYCLE** construct than for paths which are not. For normal timing checks, the hold edge pair is chosen by considering the two hold conditions with respect to all possible setup edge pairs.

For all paths affected by a **MULTI_CYCLE** construct, the default hold edge pair is chosen by considering the two hold conditions only with respect to a single setup edge pair, rather than by considering them with respect to every setup edge pair.

The following procedure is used to determine the hold edge pair:

- If the **SETUP** option is specified, then the default hold edge pair is chosen with respect to the adjusted setup edge pair. If the **HOLD** option is specified but the **SETUP** option is not, then the default hold edge pair is chosen with respect to the default setup edge pair.
- The default hold edge pair is chosen to reflect the more restrictive of the two hold conditions (the most positive difference between the target root edge and the source root edge).
- An adjusted hold edge pair is always determined, regardless of whether the **HOLD** option is specified. If the **HOLD** option is not specified, the hold *num_cycles* parameter is set to 0. If **HOLD** option is specified and the **SETUP** option is not.
  - By default, or if **SOURCE** is specified, the hold *num_cycles* parameter affects the source root edge. Instead of the default source root edge, the edge which arrives *num_cycles* cycles later is used.
  - If **TARGET** is specified, the hold *num_cycles* parameter affects the target root edge. Instead of the default target root edge, the edge which arrives *num_cycles* cycles earlier is used.
- The adjusted active edges for the hold check are the propagated versions of the adjusted root edges, measured at the source and target.

Adjustments can be made independently to the active edges of the setup check and hold check. However, the hold check root edges are defined with respect to the setup check root edges, so a setup offset will implicitly cause a change in the active edges used in the hold check.

When both a setup and hold offset are specified, the setup offset is interpreted first, establishing a new default hold edge pair. The hold offset is then applied to the edges of that pair.

If an *edge_identifier* is given, it specifies which data edge at the path target is affected by the changes in the active edges of the clock. If no edge is specified, both the rising and falling data edges at the target are affected.

Example

```
(TIMING
  (ENVIRONMENT
    (CLOCK “100 MHz 50/50” clk1)
    (CLOCK “50 MHz 50/50” divider.clkout)
  )
  (EXCEPTIONS
    (MULTI_CYCLE (SETUP 3 SOURCE) (HOLD 1) posedge
      (FROM “100 MHz 50/50”) (TO “50 MHz 50/50”)
    )
  )
)
```

The multi-cycle path specification in this example has the following effects on all paths whose source clock originates at *clk1* and whose target clock originates at *divider.clkout* (as well as any other paths with `“100 MHz 50/50”` as the source clock waveform and `“50 MHz 50/50”` as the target clock waveform):

- For the setup check on rising data edges at the target, the active edge at the source is two source clock cycles earlier than the default. The default active edge at the target is unchanged.
- The hold check on rising data edges at the target is affected by the setup adjustment as well as the hold adjustment. After applying the setup adjustment, the two hold conditions are considered with respect to the adjusted setup edge pair to determine the new default hold edge pair. This will generally cause the source edge of the default hold edge pair to be two cycles earlier than if no setup adjustment was specified.

  The hold adjustment is then applied, resulting in the hold active edge at the source being one source clock cycle later than in the default hold edge pair, while the hold active edge at the target is the same as in the default hold edge pair.
- The setup and hold checks on falling data edges at the target are unaffected by the multi-cycle specification.

Example

```
(MULTI_CYCLE (SETUP 2)
  (FROM ff1.clk) (TO ff2.d ff3.d)
)
```

The multi-cycle path specification in this example has the following effects on all paths starting at *ff1* and ending at *ff2* or *ff3*:

- For the setup check on both rising and falling data edges, the active edge at the target is one target clock cycle later than the default. The default active edge at the source is unchanged.

- The hold check on both rising and falling data edges at the target is implicitly affected by the setup adjustment. After applying the setup adjustment, the two hold conditions are considered with respect to the adjusted setup edge pair to determine the new default hold edge pair, which is used without adjustment in the hold check.

Level 1 Multi-Cycle Paths

In Level 1, the paths can be identified by an arc, a single port, or multiple ports.

Syntax

multi_cycle_spec_1 ::= **(** *label?* **MULTI_CYCLE**
multi_cycle_option+
multi_cycle_path_spec_1+ **)**
multi_cycle_path_spec_1 ::= *arc_spec*
||= *thru_spec*
||= *thru_all_spec*

Example

```
(LEVEL 1
  (MULTI_CYCLE (SETUP 3 SOURCE) (THRU and1.in1))
)
```

The multi-cycle path specification in this example has the following effects on all paths through *and1.in1*:

- For the setup check on both rising and falling data edges, the active edge at the source is three source clock cycles earlier than the default. The default active edge at the target is unchanged.
- The hold check on both rising and falling data edges at the target is implicitly affected by the setup adjustment. After applying the setup adjustment, the two hold conditions are considered with respect to the adjusted setup edge pair to determine the new default hold edge pair, which is used without adjustment in the hold check.

Example

```
(LEVEL 1
  (MULTI_CYCLE (HOLD 1 TARGET) negedge
    (THRU_ALL nor2.in1 and3.in2))
)
```

The multi-cycle path specification in this example has the following effects on all paths through both *nor2.in1* and *and3.in2*:

- The setup check on falling data edges at the target is not affected by the specification. However, this setup check does establish the default setup edge pair used by the hold check.

- The hold check on falling data edges at the target is affected by the hold adjustment. The two hold conditions are considered with respect to the default setup edge pair to determine the new default hold edge pair.

  The hold adjustment is then applied, resulting in the hold active edge at the target being one target clock cycle earlier than in the default hold edge pair, while the hold active edge at the source is the same as in the default hold edge pair.

- The setup and hold checks on rising data edges at the target are not affected by the multi-cycle specification.

Combinational Delays

The **PATH_DELAY** construct specifies constraints on the delay of paths through non-sequential parts of the design, such as the following:

- Paths through combinational logic
- Connections between hierarchical blocks
- Paths between asynchronous clock domains
- Gated clock enable signals

The **PATH_DELAY** construct describes constraints on the combinational delay through a portion of the design, while the **EXTERNAL_DELAY** construct describes purely combinational delays which are external to that portion of the design. The external delays are added to the computed path delays within that portion of the design before comparing to the path delay constraint.

The **PATH_DELAY** construct must not be used to define clock tree insertion delays. The **CLOCK_DELAY** construct must be used instead (see “Clock Delay” on page 92).

Syntax

path_delay_spec_0 ::= **(** *label?* **PATH_DELAY**
path_delay_value endpoints_spec+ **)**

path_delay_spec_1 ::= **(** *label?* **PATH_DELAY**
path_delay_value
path_delay_path_spec_1+ **)**

path_delay_value ::= **(** *timing_check waveform_edge_identifier* NUMBER **)**
||= *rise_fall_min_max*

path_delay_path_spec_1 ::= *arc_spec*
||= *thru_spec*
||= *thru_all_spec*

When the first form of *path_delay_value* is used, the **PATH_DELAY** construct can be specified multiple times for different delay constraints

which only apply to certain edges and timing checks on the same set of paths, and the union of these constraints is taken.

When a path constrained by a **PATH_DELAY** construct starts or ends at a sequential element, the combinational delay constraint for that path is implicitly adjusted to include the effect of clock skew and timing checks.

Max Transition Times

The **MAX_TRANSITION_TIME** construct specifies the constraint on the transition time of a net as measured at a specified output or bidirectional port.

Syntax

transition_time_spec ::= **(** *label?* **MAX_TRANSITION_TIME** *rise_fall port_instance** **)**

The *rise_fall* values are time values and must be specified in the units defined by the *time_scale*. If no voltage thresholds are specified for measuring the transition times (see “Voltage Threshold” on page 48), the *rise_fall* values must specify the time required to transition between the 10 and 90 percent points of the power supply voltage.

A *port_instance* must be an output or bidirectional port on a cell contained within the current GCF cell. The default transition time constraint, which can be set by omitting the *port_instances*, applies to all output pins contained within the current GCF cell.

Usually, the transition time is specified in the library. If the transition time is specified in both the library and the GCF file, the more restrictive constraint will be used.

Latch-Based Borrowing

The **BORROW_LIMIT** construct specifies the maximum amount of time that can be borrowed by one cycle from the next cycle when using level-sensitive latches. This construct is a Level 1 construct.

Data normally starts propagating from a source latch at the opening edge of the source clock. It must arrive at the target latch input before the opening edge of the target clock, thereby ensuring consistency across multiple cycles.

Time borrowing allows data to arrive at a target latch during the active portion of the target’s clock. To ensure consistency across multiple clock cycles, the delay allowed for paths starting at that latch must be reduced by the difference between the actual arrival time at the latch and the opening edge of the clock (the time borrowed by paths in the previous cycle).

The default limit on time borrowing for a given latch is the active pulse width of the clock minus the setup time of the latch. The *borrow_limit* construct can only be used to specify a smaller limit; larger limits are ignored.

Syntax

borrow_limit_spec ::= **(** *label?* **BORROW_LIMIT** NUMBER *port_instance** **)**

If no *port_instance* is specified, borrowing will be restricted for all level-sensitive latches.

If a *port_instance* that was identified as a clock (through the **CLOCK** construct—see "Clock Specifications" on page 66) is specified, borrowing will be restricted for all level-sensitive latches in the transitive fanout of that clock.

Otherwise, the *port_instances* must be clock input pins of level-sensitive latches.

Clock Delay

The **CLOCK_DELAY** construct is used to specify the following constraints:

- The insertion delay through a clock distribution network
- The skew in the insertion delay between different leaf pins of the network
- The slew of the clock at the leaf pins of the network

This construct is a Level 1 construct.

Syntax

clock_delay_spec ::= **(** *label?* **CLOCK_DELAY** *clock_root leaf_spec+* **)**

clock_root ::= *port_instance*
||= **(** *cell_instance input_port output_port* **)**

leaf_spec ::= **(** *leaf_delay_spec+ port_instance** **)**

leaf_delay_spec ::= *insertion_delay_spec*
||= *clock_skew_spec*
||= *clock_slew_spec*

insertion_delay_spec ::= **(** **INSERTION_DELAY** *rise_fall_min_max* **)**

clock_skew_spec ::= **(** **SKEW** *min_max* **)**

clock_slew_spec ::= **(** **SLEW** *slew_value* **)**

If a *port_instance* is specified for the clock root, it indicates the pin that is the source of the clock distribution network. Insertion delay and skew are

measured from that pin to each of the leaf *port_instances*.

If a *cell_instance* is specified for the clock root, it gives the instance name of a cell that drives the clock distribution network. Insertion delay is measured from the specified *input_port* through the *output_port* to each of the leaf *port_instances*.

Insertion delay, skew, and slew can be specified.

If no leaf *port_instance* is specified, the *leaf_spec* applies to all primitive clock input pins that are reached by tracing forward from the specified *clock_root* through combinational logic. These primitive clock input pins are the implicit leaf *port_instances*.

The slew values at a primitive clock input pin will be determined using the following precedence order:

- The slew specified explicitly by an **INTERNAL_SLEW** construct
- The calculated slew, when the physical clock network has already been implemented
- The slew specified by a **CLOCK_DELAY** construct that specifically lists the leaf pin
- The slew specified by a **CLOCK_DELAY** construct that includes the pin as an implicit leaf *port_instance*
- If several **CLOCK_DELAY** constructs with slew specifications implicitly include the pin, the slew is taken from the specification where the root is closest to the leaf pin.
- The default **INTERNAL_SLEW**
- The default **INPUT_SLEW**
- 0

Timing Exception Cases

The timing exceptions can be case-dependent.

Syntax

timing_exception_case ::= **(CASE** IDENTIFIER *timing_exception_case_spec*+ **)**

timing_exception_case_spec ::= *timing_exception_spec_0*
||= *timing_exception_no_case_1*

Example

```
(EXCEPTIONS
  (level 1
    (case normal
      (multi_cycle (setup 4) (from reg1))
    )
    (case throttled
      (multi_cycle (setup 2) (from reg1))
    )

  )
)
```

In this example, the number of cycles required for paths starting at *reg1* depends on whether the clock provided to the chip is being throttled.

5

Parasitics Subset

Parasitics Subset Header

Parasitics Environment

Parasitics Constraints

Parasitics Subset Header

The parasitics subset of each cell entry in the GCF file includes the following:

- Information about the parasitics in the environment in which the cell is intended to operate
- Constraints on the parasitics within the cell

This chapter describes the parasitic environment and parasitic constraints. For information on other constructs, refer to “Extensions” on page 37, “Meta Data” on page 40, and “Include Files” on page 42.

Syntax

parasitics_subset ::= **(SUBSET PARASITICS** *parasitics_subset_body* **)**

parasitics_subset_body ::= *parasitics_subset_spec+*
||= *include*

parasitics_subset_spec ::= *parasitics_environment*
||= *parasitics_constraints*
||= *extension*
||= *meta_data*

Example

```
(CELL
  (CELLTYPE "WORKLIB" "ALU")
  (INSTANCE *)
  (SUBSET PARASITICS
    (ENVIRONMENT
       . . .
    )
    (CONSTRAINTS
       . . .
    )
  )
)
```

Parasitics Environment

The parasitics environment of a cell describes a number of conditions external to the cell that affect its timing behavior. This version of GCF includes only the external capacitance on nets connected to the cell interface pins.

Syntax

parasitics_environment ::= **(ENVIRONMENT** *parasitics_env_spec+* **)**

parasitics_env_spec ::= *parasitics_env_spec_0*
||= *parasitics_env_spec_1*

parasitics_env_spec_0 ::= *external_load_spec*
||= *extension*
||= *meta_data*

parasitics_env_spec_1 ::= **(LEVEL** 1 *parasitics_env_1+* **)**

parasitics_env_1 ::= *parasitics_env_no_case_1*
||= *parasitics_env_case*

parasitics_env_no_case_1 ::= *external_fanout_spec*

The following sections describe external loading, external fanout, and parasitic environment cases.

External Loading

The external capacitance on an interface net can be specified in terms of the actual capacitance value using the **EXTERNAL_LOAD** construct.

Syntax

external_load_spec ::= **(** *label?* **EXTERNAL_LOAD** *capacitance* *port_instance** **)**

capacitance ::= *min_max*

The capacitance can be specified for both input and output ports. If no *port_instance* is specified, the specification applies by default to all primary ports.

External Fanout

The external capacitance on an interface net can be specified in terms of the number of loads using the **EXTERNAL_FANOUT** construct. This construct is a Level 1 construct because it requires wire load models for proper interpretation.

Syntax

external_fanout_spec ::= **(** *label?* **EXTERNAL_FANOUT** *num_loads* *port_instance** **)**

num_loads ::= *min_max*

The number of external fanouts can be specified for both input and output ports. If no *port_instance* is specified, the specification applies by default to all primary ports.

Parasitics Environment Cases

The parasitics environment can be case-dependent.

Syntax

parasitics_env_case ::= **(CASE** IDENTIFIER
parasitics_env_case_spec+ **)**

parasitics_env_case_spec::= *parasitics_env_spec_0*
||= *parasitics_env_no_case_1*

Example

```
(environment
  (level 1
    (case board
      (external_load 50.0 out1)
    )
    (case tester
      (external_load 100.0 out1)
    )
  )
)
```

In this example, the external capacitance on pin *out1* depends on whether the chip is mounted on the board or whether it is being tested.

Parasitics Constraints

This version of GCF includes only the **parasitics** constraints on the nets within a cell. Two forms of constraints are currently supported. The constraint form depends on whether the net is connected to a primary port on the cell.

Syntax

parasitics_constraints ::= **(CONSTRAINTS** *parasitics_constraint+* **)**

parasitics_constraint ::= *parasitics_cnstr_spec_0*
||= *parasitics_cnstr_spec_1*

parasitics_cnstr_spec_0 ::= *internal_load_spec*
||= *load_spec*
||= *extension*
||= *meta_data*

parasitics_cnstr_spec_1 ::= **(LEVEL** 1 *parasitics_cnstr_1+* **)**

parasitics_cnstr_1 ::= *parasitics_cnstr_no_case_1*
||= *parasitics_cnstr_case*

parasitics_cnstr_no_case_1 ::= *internal_fanout_spec*
||= *fanout_spec*

The following sections describe internal loading, loading, internal fanout, fanout, and parasitic constraint cases.

Internal Loading

The constraint on the capacitance of an internal net can be specified in terms of an explicit capacitance value using the **INTERNAL_LOAD** construct.

Syntax

internal_load_spec ::= **(** *label?* **INTERNAL_LOAD** *capacitance*
*port_instance** **)**

The constraint on the capacitance of an internal net can be specified for both input and output ports. If no *port_instance* is specified, the specification applies by default to all primary ports.

Loading

The constraint on the capacitance of an internal net that is not connected to a primary port on the cell can be specified in terms of an explicit capacitance value using the **LOAD** construct.

Syntax

load_spec ::= **(** *label?* **LOAD** *capacitance*
*port_instance** **)**

The constraint on the capacitance of an internal net can be specified on any port connected to the net. If different constraints are specified on several ports connected to the same net, the most restrictive constraint will be used. If no *port_instance* is specified, the specification applies by default to all internal nets.

Internal Fanout

The constraint on the capacitance of an internal net can be specified in terms of the number of loads using the **INTERNAL_FANOUT** construct. This construct is a Level 1 construct because it requires wire load models for proper interpretation.

Syntax

internal_fanout_spec ::= **(** *label?* **INTERNAL_FANOUT** *num_loads* *port_instance** **)**

The number of internal fanouts can be specified for both input and output ports. If no *port_instance* is specified, the specification applies by default to all primary ports.

Fanout

The constraint on the capacitance of an internal net that is not connected to a primary port on the cell can be specified in terms of the number of loads using the **FANOUT** construct. This construct is a Level 1 construct because it requires wire load models for proper interpretation.

Syntax

fanout_spec ::= **(** *label?* **FANOUT** *num_loads* *port_instance** **)**

The number of fanouts can be specified on any port connected to the net. If different constraints are specified on several ports connected to the same net, the most restrictive constraint will be used. If no *port_instance* is specified, the specification applies by default to all internal nets.

Parasitics Constraint Cases

The parasitics constraints can be case-dependent, although it usually makes sense to specify the tightest constraint across all of the cases instead.

Syntax

parasitics_cnstr_case ::= **(** **CASE** IDENTIFIER *parasitics_cnstr_case_spec+* **)**

parasitics_cnstr_case_spec ::= *parasitics_cnstr_spec_0*
||= *parasitics_cnstr_no_case_1*

6

Area Subset

Area Subset Header

Area Constraints

Area Subset Header

The area subset of each cell entry in the GCF file includes the following:

- Constraints on the area of the cell
- Constraints on the area of the primitives instantiated within the cell

This chapter describes the primitive area constraints, total area constraints, cell porosity, and area constraint cases. For information on other constructs, refer to “Extensions” on page 37, “Meta Data” on page 40, and “Include Files” on page 42.

Syntax

area_subset ::= **(SUBSET AREA** *area_subset_body* **)**

area_subset_body ::= *area_cnstr_spec+*
||= *include*

area_cnstr_spec ::= *area_cnstr_spec_0*
||= *area_cnstr_spec_1*

area_cnstr_spec_0 ::= *primitive_area_spec*
||= *total_area_spec*
||= *extension*
||= *meta_data*

area_cnstr_spec_1 ::= **(LEVEL 1** *area_cnstr_1+* **)**

area_cnstr_1 ::= *area_cnstr_no_case_1*
||= *area_cnstr_case*

area_cnstr_no_case_1 ::= *porosity_spec*

Example

```
(CELL
  (CELLTYPE "WORKLIB" "ALU")
  (INSTANCE *)
  (SUBSET AREA
    (PRIMITIVE_AREA 5000)
    (TOTAL_AREA 5500)
  )
)
```

Area Constraints

Primitive Area

The cumulative area of the leaf-level primitive cells that are instantiated either directly within a cell or within its descendents can be specified using the **PRIMITIVE_AREA** construct. The primitive area does not include any physical overhead such as routing and power distribution which affect the total area of the cell.

Syntax

primitive_area_spec ::= **(** *label?* **PRIMITIVE_AREA** *area_value* **)**

area_value ::= *min_max*

If a single value is specified, it represents the maximum. If two values are specified, they represent the minimum and maximum values, respectively.

Example

```
(PRIMITIVE_AREA 5000)
```

Assuming that the *area_scale* is set so that area values in the GCF file(s) are specified in square microns, the example specifies that the total primitive area within the current cell must be less than or equal to 5000 square microns.

Total Area

The total area of a cell (including physical overhead) can be specified using the **TOTAL_AREA** construct.

Syntax

total_area_spec ::= **(** *label?* **TOTAL_AREA** *area_value* **)**

If a single value is specified, it represents the maximum. If two values are specified, they represent the minimum and maximum values, respectively.

Example

```
(TOTAL_AREA 5500)
```

Assuming that the area_scale is set so that area values in the GCF file(s) are specified in square microns, this example specifies that the total area of the current cell must be less than or equal to 5500 square microns.

Porosity

The **POROSITY** construct is a Level 1 construct and specifies the porosity of a cell.

Porosity is the percentage of the total primitive area that is available for over-the-cell routing. The total primitive area is the sum across all of the

leaf-level primitive cells which are instantiated either directly within the current cell or within its descendents.

Syntax

porosity_spec ::= **(** *label?* **POROSITY** *porosity_value* **)**
porosity_value ::= *min_max*

If a single value is specified, it represents the minimum. If two values are specified, they represent the minimum and maximum values, respectively.

Example

```
(POROSITY 40)
```

In this example, at least 40 percent of the primitive area within the current cell must be available for over-the-cell routing.

Area Constraint Cases

The area constraints can be case-dependent, although it usually makes sense to specify the tightest constraint across all of the cases instead.

Syntax

area_cnstr_case ::= **(** **CASE** IDENTIFIER *area_cnstr_case_spec+* **)**
area_cnstr_case_spec ::= *area_cnstr_spec_0*
||= *area_cnstr_no_case_1*

7

Power Subset

Power Subset Header

Power Constraints

Power Subset Header

The power subset of each cell entry in the GCF file includes the following:

- Constraints on the average power consumed by the cell and the primitives instantiated within it
- Constraints on the power consumed by particular nets

This chapter describes the average cell power constraints, average net power constraints, and power constraint cases. For information on other constructs, refer to “Extensions” on page 37, “Meta Data” on page 40, and “Include Files” on page 42.

Syntax

power_subset ::= **(SUBSET POWER** *power_subset_body* **)**
power_subset_body ::= *power_cnstr_spec+*
||= *include*
power_cnstr_spec ::= *power_cnstr_spec_0*
||= *power_cnstr_spec_1*
power_cnstr_spec_0 ::= *average_cell_power*
||= *average_net_power*
||= *extension*
||= *meta_data*
power_cnstr_spec_1 ::= **(LEVEL 1** *power_cnstr_1+* **)**
power_cnstr_1 ::= *power_cnstr_case*

Example

```
(CELL
  (CELLTYPE "WORKLIB" "ALU")
  (INSTANCE *)
  (SUBSET POWER
    (AVG_CELL_POWER 50)
  )
)
```

Power Constraints

Average Cell Power

The average power consumed by the current GCF cell instance can be specified using the **AVG_CELL_POWER** construct.

Syntax

average_cell_power ::= **(** *label?* **AVG_CELL_POWER** *power_value* **)**

power_value ::= *min_max*

If a single value is specified, it represents the maximum. If two values are specified, they represent the minimum and the maximum values, respectively.

Example

```
(AVG_CELL_POWER 50.0)
```

Assuming that the *power_scale* is set so that power values in the GCF file(s) are specified in milliwatts, the example specifies that the average power consumed by the current cell instance must be less than or equal to 50 milliwatts.

Average Net Power

The average power dissipated by the capacitance in a net can be specified using the **AVG_NET_POWER** construct. This construct is generally only used for clock nets.

Syntax

average_net_power ::= **(** *label?* **AVG_NET_POWER** *power_value* *port_instance* **)**

The power is specified for the physical net as a whole, although the net is identified using one of the *port_instances* connected to the net.

Example

```
(AVG_NET_POWER 1000.0 CLKBUF.OUT)
```

Assuming that the *power_scale* is set so that power values in the GCF file(s) are specified in milliwatts, the example specifies that the average power consumed by the specified net must be less than or equal to 1 watt.

Power Constraint Cases

The power constraints can be case-dependent, although it usually makes sense to specify the tightest constraint across all of the cases instead.

Syntax

power_cnstr_case ::= **(CASE** IDENTIFIER *power_cnstr_case_spec*+ **)**

power_cnstr_case_spec ::= *power_cnstr_spec_0*

8

Syntax of GCF

GCF File Characters

Syntax Conventions

GCF File Syntax

GCF File Characters

The legal GCF character set and the method of including comments in GCF files are described in this section.

GCF Characters

The characters you can use in an GCF file are the following:

- Alphanumeric characters – the letters of the alphabet, all the numbers, and the underscore ‘_’ character.
- Special characters – any character other than alphanumeric characters (which includes the underscore as defined above) is a special character. The following is a list of special characters:
  ! " # $ % & ´ ( ) * + , - . / : ; < = > ? @ [\ ] ^ ` { | } ~
- Syntax characters – these are special characters required by the syntax. Examples are: ( ) " * : [ ] ? and the hierarchy delimiter character but see also the definitions of GCF operators, etc.
- The escape character – to use any special character in an IDENTIFIER, prefix it with the escape character, a backslash ‘\’. This includes the backslash character itself: two consecutive backslashes are used to represent a single backslash in the original IDENTIFIER.

  See “Variables” on page 119 for a description of an IDENTIFIER. Note that if the character would normally have any special meaning in an IDENTIFIER, this is lost when the character is escaped.
- Hierarchy delimiter character – either the period ‘.’ or the slash ‘/’ can be established as the hierarchy delimiter character. This character only has this special meaning in an IDENTIFIER. An escaped hierarchy delimiter character loses its meaning as a hierarchy delimiter.
- Left index delimiter character - the left bracket ‘[‘, left parenthesis ‘(‘, or left angle bracket ‘<‘ can be established as the left index delimiter character. The left index delimiter is used as the first delimiter in a bit-spec. This character only has this special meaning in an IDENTIFIER. used as the name of a port or cell instance. An escaped left index delimiter character loses its meaning as a left index delimiter.
- Right index delimiter character - the right bracket ‘]‘, right parenthesis ‘)‘, or right angle bracket ‘>’ can be established as the right index delimiter character. The right index delimiter is used as the last delimiter in a bit-spec. This character only has this special meaning in an IDENTIFIER used as the name of a port or cell instance. An escaped right index delimiter character loses its meaning as a right index delimiter.

- White space characters – tabs, spaces and newlines are considered white space. Use white space to separate lexical tokens.

Keywords, IDENTIFIERs, characters, and numbers must be delimited either by syntax characters or by white space.

Comments

Comments can be placed in GCF files using either “C” or “C++” styles.

“C”-style comments begin with /* and end with */. Nesting of “C”-style comments is not permitted. The places in an GCF file where it is legal to put “C”-style comments are not defined by this specification. Different annotators can have different capabilities in this regard.

“C++”-style comments begin with // and continue until the end of the current line (the next newline character). Annotators should ignore the double-slash and any text after them on any line in the file.

Syntax Conventions

Notation

The notation used in presenting the syntax of GCF are as follows:

item	*item* is a symbol for a syntax construct item.
item ::= *definition*	the BNF symbol *item* is defined as *definition.*
item ::= *definition*1 ||= *definition*2	the BNF symbol *item* is defined either as *definition1* or as *definition2.* (any number of alternative syntax definitions can appear)
item?	*item* is optional in the definition (it can appear once or not at all).
*item**	*item* can appear zero or any number of times.
item+	*item* can appear one or more times (but cannot be omitted).
KEYWORD	is a keyword and appears in the file as shown. Keywords are shown in uppercase bold for easy identification but are case insensitive.
VARIABLE	is a symbol for a variable. Variable symbols are shown in uppercase for easy identification. Some variables are defined as one of a number of discrete choices (e.g. HCHAR, which is either a period or a slash). Other variables represent user data such as names and numbers.

Variables

This section defines the user data variables used in GCF. Variables which must be one of a number of choices (enumerations) are defined in the main syntax definition which follows.

QSTRING	is a string of any legal GCF characters and spaces, excluding tabs and newlines, enclosed by double-quotes. Except for the double-quote itself, special characters lose their special meaning in a QSTRING. To embed a double-quote within a QSTRING, escape it with a backslash.
NUMBER	is a non-negative (zero or positive) real number, for example: 0, 1, 0.0, 3.4, .7, 0.3, 2.4e2, 5.3e-1, 8.2E+5
RNUMBER	is a positive, zero or negative real number, for example: 0, 1, 0.0, -3.4, .7, -0.3, 2.4e2, -5.3e-1, 8.2E+5
DNUMBER	is a non-negative integer number, for example: +12, 23, 0
INUMBER	is an integer number, for example: -5, 10, 0, +7
IDENTIFIER	is the name of an object in the design. This could be an instance of a design block or cell or a port depending on where the IDENTIFIER occurs in the GCF file. Identifiers can be up to 1024 characters long.

The following characters can be used in an identifier:

- Alphanumeric characters – the letters of the alphabet, all the numbers, and the underscore '_' character. IDENTIFIERs are case-sensitive, i.e. uppercase and lowercase letters are considered different.
- Bit specs – to indicate an object selected from an array of objects, for example a single port selected from a bus port or an instance from an array of instances, use a "bit spec" at the end of the IDENTIFIER of the array (with no separating white space). A bit spec consists of the left and right index delimiters ('[' and ']', by default) enclosing a range.

  To select a single object, the range should be a single positive integer, for example, [4].

  To select a contiguous group of objects, the range should be a pair of positive integers separated by a colon (':'), for example, [3:31] and [15:0].

  To select all objects in the array, the range should be the WILDCARD, an asterisk ('*'). For example, [*].
- Hierarchy delimiter character – see "PATH" below.
- The escape character '\' – if you want to use a non-alphanumeric character as a part of an IDENTIFIER it must be escaped by being prefixed with the '\' character. Examples are shown below.
  Note – this escaping mechanism is different from Verilog HDL where the entire IDENTIFIER is escaped by placing one escape character (\) before the IDENTIFIER and a white space after the IDENTIFIER.
  Characters that have special meaning in identifiers, such as the left and right index delimiters and the hierarchy delimiter, lose that special meaning when escaped.
- Do not use white space (spaces, tabs or newlines) in an IDENTIFIER.

Examples of correct IDENTIFIERs are:

```
AMUX\+BMUX

Cache_Row_\#4

mem_array\[0\:1023\]\(0\:15\)   ; From a language where square
                                ; brackets indicates arrays
                                ; parentheses indicates bit specs

pipe4\-done\&enb[3]             ; Unescaped square brackets
                                ; represent a bit spec
```

PATH	is a hierarchical IDENTIFIER. The names of levels in the design hierarchy must be separated by the hierarchy delimiter character. A path is always interpreted relative to a particular region of the design (which can be the top level cell in the design, so a leading hierarchy delimiter character should not be used. The hierarchy delimiter character must not be escaped or it loses its meaning as a hierarchy delimiter. See “Delimiters” on page 30 for details on how the hierarchy delimiter character is established.
PARTIAL_PATH	is either an IDENTIFIER or a PATH. A partial path is used in combination with a *prefix_id* to reduce the file size when many PATHs contain a common prefix. See “Design References” on page 56 for details on how a *prefix_id* is established.
HCHAR	is the hierarchy delimiter character.
LI_CHAR	is the left index delimiter character.
RI_CHAR	is the right index delimiter character.
COLON	is the colon character (‘:’).
WILDCARD	is the asterisk character (‘*’).

GCF File Syntax

The formal syntax definition for the General Constraint Format is given here. It is not possible, using the notation chosen, to clearly show how white-space must be used in the GCF file. Some explanations and comments are included in the formal descriptions. A double-slash (//) indicates comments which are not part of the syntax definition.

constraint_file ::= **(GCF** *header section+* **)**

header ::= **(HEADER** *version header_info** **)**

section ::= *globals*
||= *cell_spec*
||= *extension*
||= *meta_data*
||= *include*

version ::= **(VERSION** QSTRING **)**

header_info ::= *design_name*
||= *date*
||= *program*
||= *delimiters*
||= *time_scale*
||= *cap_scale*
||= *res_scale*
||= *length_scale*
||= *area_scale*
||= *voltage_scale*
||= *power_scale*
||= *extension*

design_name ::= **(DESIGN** QSTRING **)**

date ::= **(DATE** QSTRING **)**

program ::= **(PROGRAM** *program_name program_version program_company* **)**

program_name ::= QSTRING
program_version ::= QSTRING
program_company ::= QSTRING

delimiters ::= **(DELIMITERS** QSTRING **)**

time_scale ::= **(TIME_SCALE** *multiplier* **)**
cap_scale ::= **(CAP_SCALE** *multiplier* **)**
res_scale ::= **(RES_SCALE** *multiplier* **)**
length_scale ::= **(LENGTH_SCALE** *multiplier* **)**
area_scale ::= **(AREA_SCALE** *multiplier* **)**
voltage_scale ::= **(VOLTAGE_SCALE** *multiplier* **)**
power_scale ::= **(POWER_SCALE** *multiplier* **)**

multiplier ::= NUMBER

Extensions

Extensions are defined as follows:

extension ::= **(EXTENSION** QSTRING *extension_construct+* **)**

extension_construct ::= **(** *user_defined* **)**
||= *include*

Labels

Constraint labels are defined as follows:

label ::= *label_id* COLON

label_id ::= IDENTIFIER
||= QSTRING

Meta Data

Meta data specifications are defined as follows:

meta_data ::= **(LEVEL 1** *meta_data_1+* **)**

meta_data_1 ::= **(META** *meta_construct+* **)**

meta_construct ::= *precedence*
||= *meta_reserved*
||= *include*

precedence ::= **(PRECEDENCE (** *label_id label_id+***))**

meta_reserved ::= **(** IDENTIFIER *reserved_for_future_definition* **)**

Include Specifications

Include specifications are defined as follows:

include ::= **(INCLUDE** QSTRING **)**

Value Types

Common types of values used in many constraints are defined as follows:

min_and_max ::= NUMBER NUMBER

r_min_and_max ::= RNUMBER RNUMBER

min_max ::= NUMBER NUMBER?

r_min_max ::= RNUMBER RNUMBER?

rise_fall_min_max ::= NUMBER
||= NUMBER NUMBER
||= NUMBER NUMBER NUMBER NUMBER

r_rise_fall_min_max ::= RNUMBER
||= RNUMBER RNUMBER
||= RNUMBER RNUMBER RNUMBER RNUMBER

rise_and_fall ::= NUMBER NUMBER

r_rise_and_fall ::= RNUMBER RNUMBER

rise_fall ::= NUMBER NUMBER?

r_rise_fall ::= RNUMBER RNUMBER?

Globals

The globals section is defined as follows:

globals ::= **(GLOBALS** *globals_subset+* **)**

globals_subset ::= *env_globals_subset*
||= *timing_globals_subset*
||= *extension*
||= *meta_data*

Environment Globals

The environment globals are defined as follows:

env_globals_subset ::= **(GLOBALS_SUBSET ENVIRONMENT** *env_globals_body* **)**

env_globals_body ::= *env_globals_spec+*
||= *include*

env_globals_spec ::= *env_globals_spec_0*
||= *env_globals_spec_1*

env_globals_spec_0 ::= *process*
||= *voltage*
||= *temperature*
||= *operating_conditions*
||= *voltage_threshold*
||= *extension*
||= *meta_data*

process ::= **(PROCESS** *min_and_max* **)**

voltage ::= **(VOLTAGE** *r_min_and_max* **)**

temperature ::= **(TEMPERATURE** *r_min_and_max* **)**

operating_conditions ::= **(** *label?* **OPERATING_CONDITIONS**
QSTRING
process_value
voltage_value
temperature_value **)**

process_value ::= NUMBER

voltage_value ::= RNUMBER

temperature_value ::= RNUMBER

voltage_threshold ::= **(** *label?* **VOLTAGE_THRESHOLD**
min_and_max **)**

env_globals_spec_1 ::= **(LEVEL** 1 *env_globals_1+* **)**

env_globals_1 ::= *env_globals_case*

env_globals_case ::= **(CASE** IDENTIFIER *env_globals_case_spec+* **)**

env_globals_case_spec ::= *env_globals_spec_0*

Timing Globals

The timing globals are defined as follows:

timing_globals_subset ::= **(GLOBALS_SUBSET TIMING** *timing_globals_body* **)**

timing_globals_body ::= *timing_globals_spec+*
||= *include*

timing_globals_spec ::= *timing_globals_spec_0*
||= *timing_globals_spec_1*

timing_globals_spec_0 ::= *primary_waveform*
||= *extension*
||= *meta_data*

primary_waveform ::= **(** *label?* **WAVEFORM** *waveform_name period edge_pair_list* **)**

waveform_name ::= QSTRING

period ::= NUMBER

edge_pair_list ::= *pos_pair+*
||= *neg_pair+*

pos_pair ::= *pos_edge neg_edge*

neg_pair ::= *neg_edge pos_edge*

pos_edge ::= **(POSEDGE** *min_max* **)**

neg_edge ::= **(NEGEDGE** *min_max* **)**

timing_globals_spec_1 ::= **(LEVEL** 1 *timing_globals_1+* **)**

timing_globals_1 ::= *timing_globals_no_case_1*
||= *timing_globals_case*

timing_globals_no_case_1 ::= *derived_waveform*
||= *clock_group*

derived_waveform ::= **(** *label*? **DERIVED_WAVEFORM**
waveform_name parent_waveform_name
derived_waveform_option+ **)**

parent_waveform_name ::= QSTRING

derived_waveform_option ::= *period_multiplier*
||= *phase_shift*
||= *skew_adjustment*

period_multiplier ::= **(PERIOD_MULTIPLIER** DNUMBER **)**

phase_shift ::= **(PHASE_SHIFT** RNUMBER **)**

skew_adjustment ::= **(SKEW_ADJUSTMENT** *edge_pair_list* **)**

clock_group ::= **(** *label*? **CLOCK_GROUP** *clock_group_name waveform_name*+ **)**

clock_group_name ::= QSTRING

timing_globals_case ::= **(CASE** IDENTIFIER *timing_globals_case_spec*+ **)**

timing_globals_case_spec ::= *timing_globals_spec_0*
||= *timing_globals_no_case_1*

Design References

The references to design elements are defined as follows:

name_prefixes ::= **(NAME_PREFIXES** *num_prefixes name_prefix+*)

num_prefixes ::= DNUMBER

name_prefix ::= *prefix_id* QSTRING

prefix_id ::= DNUMBER

cell_instance ::= PATH
||= **(** *prefix_id* **)**
||= **(** *prefix_id* PARTIAL_PATH **)**

port_instance ::= *port*
||= PATH HCHAR *port*
||= **(** *prefix_id port* **)**
||= **(** *prefix_id* PARTIAL_PATH HCHAR *port* **)**

/* There should be no white space separating the PATH or PARTIAL_PATH, HCHAR, and *port* components of a *port_instance* */

port ::= *scalar_port*
||= *bus_port*

input_port ::= *scalar_port*

output_port ::= *scalar_port*

scalar_port ::= IDENTIFIER
||= IDENTIFIER LI_CHAR DNUMBER RI_CHAR

bus_port ::= IDENTIFIER LI_CHAR DNUMBER COLON DNUMBER RI_CHAR
||= IDENTIFIER LI_CHAR WILDCARD RI_CHAR

cell_id ::= **(CELLTYPE** *cell_name* **)**
||= **(CELLTYPE** *library_name cell_name view_name?* **)**

cell_name ::= QSTRING

library_name ::= QSTRING

view_name ::= QSTRING

Cell Entries

Cell entries are defined as follows:

cell_spec ::= **(CELL** *cell_instance_spec cell_body_spec+* **)**

cell_instance_spec ::= *cell_instance_path*
||= **(** *cell_instance_path+* **)**
||= **( )**
||= *cell_views*

cell_instance_path ::= PATH

cell_views ::= **(CELLTYPE** *cell_name* **)**
||= **(CELLTYPE** *library_name cell_name view_name** **)**

cell_body_spec ::= *name_prefixes*
||= *subset*
||= *extension*
||= *meta_data*
||= *include*

Subsets

Subset specifications are defined as follows:

subset ::= *timing_subset*
||= *parasitics_subset*
||= *area_subset*
||= *power_subset*

Timing Subset

The timing subset is defined as follows:

timing_subset ::= **(SUBSET TIMING** *timing_subset_body* **)**

timing_subset_body ::= *timing_subset_spec+*
||= *include*

timing_subset_spec ::= *timing_environment*
||= *timing_exceptions*
||= *extension*
||= *meta_data*

Timing Environment

The timing environment is defined as follows:

timing_environment ::= **(ENVIRONMENT** *timing_env_spec+* **)**

timing_env_spec ::= *timing_env_spec_0*
||= *timing_env_spec_1*

timing_env_spec_0 ::= *clock_spec*
||= *arrival_spec*
||= *departure_spec*
||= *external_delay_spec*
||= *driver_spec*
||= *input_slew_spec*
||= *extension*
||= *meta_data*

clock_spec ::= **(** *label*? **CLOCK** *waveform_name port_instance+* **)**

arrival_spec ::= **(** *label*? **ARRIVAL** *waveform_edge arrival_value port_instance** **)**

arrival_value ::= **(** *waveform_edge_identifier r_min_max* **)**
||= *r_rise_fall_min_max*

departure_spec ::= **(** *label*? **DEPARTURE** *waveform_edge departure_value port_instance** **)**

departure_value ::= *setup_rise_fall hold_rise_fall*
||= **(** *waveform_edge_identifier setup_value hold_value* **)**

setup_rise_fall ::= *r_rise_and_fall*
hold_rise_fall ::= *r_rise_and_fall*
setup_value ::= RNUMBER
hold_value ::= RNUMBER

external_delay_spec ::= **(** *label?* **EXTERNAL_DELAY**
external_delay_value endpoints_spec+ **)**

external_delay_value ::= **(** *waveform_edge_identifier r_min_max* **)**
||= *r_rise_fall_min_max*

waveform_edge ::= **(** *waveform_edge_identifier waveform_name* **)**

driver_spec ::= *driver_cell_spec*
||= *driver_strength_spec*

driver_cell_spec ::= **(** *label?* **DRIVER_CELL**
driver_cell_port_spec
driver_cell_options?
opt_port_instance_list **)**

driver_cell_port_spec ::= **(** *cell_id* **)**
||= **(** *cell_id output_port* **)**
||= **(** *cell_id input_port output_port* **)**

driver_cell_options ::= **(** *driver_cell_option+* **)**

driver_cell_option ::= *drive_multiplier*
||= *driver_input_slew*
||= *waveform_edge_identifier*

drive_multiplier ::= **(** **PARALLEL_DRIVERS** DNUMBER **)**

driver_input_slew ::= **(** **INPUT_SLEW** *slew_value input_port** **)**

slew_value ::= *rise_fall_min_max*

driver_strength_spec ::= **(** *label*? **DRIVER_STRENGTH** *strength_value port_instance** **)**

strength_value ::= *rise_fall*

input_slew_spec ::= **(** *label*? **INPUT_SLEW** *slew_value port_instance** **)**

timing_env_spec_1 ::= **(** **LEVEL** 1 *timing_env_1+* **)**

timing_env_1 ::= *timing_env_no_case_1*
||= *timing_env_case*

timing_env_no_case_1 ::= *constant_spec*
||= *operating_conditions*
||= *internal_slew_spec*

constant_spec ::= **(** *label*? **CONSTANT** *constant_value port_instance+***)**

constant_value ::= **0**
||= **1**

internal_slew_spec ::= **(** *label*? **INTERNAL_SLEW** *slew_value port_instance** **)**

timing_env_case ::= **(CASE** IDENTIFIER *timing_env_case_spec*+ **)**

timing_env_case_spec ::= *timing_env_spec_0*
||= *timing_env_no_case_1*

Timing Exceptions

The timing exceptions are defined as follows:

timing_exceptions ::= **(EXCEPTIONS** *timing_exception_spec+* **)**

timing_exception_spec ::= *timing_exception_spec_0*
||= *timing_exception_spec_1*

timing_exception_spec_0 ::= *disable_spec_0*
||= *multi_cycle_spec_0*
||= *path_delay_spec_0*
||= *transition_time_spec*
||= *extension*
||= *meta_data*

timing_exception_spec_1 ::= **(LEVEL** 1 *timing_exception_1+* **)**

timing_exception_1 ::= *timing_exception_no_case_1*
||= *timing_exception_case*

timing_exception_no_case_1 ::= *disable_spec_1*
||= *multi_cycle_spec_1*
||= *path_delay_spec_1*
||= *internal_slew_spec*
||= *borrow_limit_spec*
||= *clock_delay_spec*

timing_exception_case ::= **(CASE** IDENTIFIER *timing_exception_case_spec+* **)**

timing_exception_case_spec ::= *timing_exception_spec_0*
||= *timing_exception_no_case_1*

arc_spec ::= **(ARC** *port_instance port_instance* **)**

endpoints_spec ::= *from_spec*
||= *to_spec*
||= **(** *from_spec to_spec* **)**

from_spec ::= **(FROM** *from_to_item+* **)**

to_spec ::= **(TO** *from_to_item+* **)**

from_to_item ::= *port_instance*
||= *cell_instance*
||= *waveform_name*

thru_spec ::= **(THRU** *port_instance* **)**

thru_all_spec ::= **(THRU_ALL** *port_instance port_instance+* **)**

disable_spec_0 ::= *disable_item_spec_0*
||= *disable_endpoints_spec_0*

disable_item_spec_0 ::= *label?* **DISABLE** *disable_item_0+* **)**

disable_item_0 ::= *port_instance*
||= *cell_instance*
||= *arc_spec*

disable_endpoints_spec_0 ::= **(** *label?* **DISABLE** *endpoints_spec+ disable_endpoints_options?* **)**

disable_endpoints_options ::= *timing_check*
||= *edge_identifier*
||= *timing_check edge_identifier*

timing_check ::= **SETUP**
||= **HOLD**

disable_spec_1 ::= *disable_edges_spec_1*
||= *disable_cell_spec_1*

disable_edges_spec_1 ::= **(** *label?* **DISABLE** *disable_edges_path_spec+ timing_check?* **)**

disable_cell_spec_1 ::= **(** *label?* **DISABLE** *disable_cell_path_spec+* **)**

disable_edges_path_spec ::= *thru_edge_spec*
||= *arc_edges_spec*
||= *thru_all_edges_spec*

thru_edge_spec ::= **(** **THRU** *port_instance_edge* **)**

arc_edges_spec ::= **(** **ARC** *port_instance_edge port_instance_edge* **)**

thru_all_edges_spec ::= **(** **THRU_ALL** *port_instance_edge port_instance_edge+* **)**

port_instance_edge ::= **(** *edge_identifier port_instance* **)**

disable_cell_path_spec ::= *disable_instance_spec*
||= *disable_master_spec*

disable_instance_spec ::= **(** **INSTANCE** *cell_instance+* **)**

disable_master_spec ::= **(** **MASTER** *cell_id* **)**

multi_cycle_spec_0 ::= **(** *label*? **MULTI_CYCLE** *multi_cycle_option+ endpoints_spec+* **)**

multi_cycle_option ::= *timing_check_offset*
||= *edge_identifier*

timing_check_offset ::= **(** *timing_check num_cycles reference_clock?* **)**

reference_clock ::= **SOURCE**
||= **TARGET**

num_cycles ::= INUMBER

multi_cycle_spec_1 ::= **(** *label*? **MULTI_CYCLE** *multi_cycle_option+ multi_cycle_path_spec_1+* **)**

multi_cycle_path_spec_1 ::= *arc_spec*
||= *thru_spec*
||= *thru_all_spec*

path_delay_spec_0 ::= **(** *label*? **PATH_DELAY** *path_selay_value endpoints_spec+* **)**

path_delay_spec_1 ::= **(** *label*? **PATH_DELAY**
path_delay_value path_delay_path_spec_1+ **)**

path_delay_value ::= **(** *timing_check waveform_edge_identifier* NUMBER **)**
||= *rise_fall_min_max*

path_delay_path_spec_1 ::= *arc_spec*
||= *thru_spec*
||= *thru_all_spec*

transition_time_spec ::= **(** *label*? **MAX_TRANSITION_TIME** *rise_fall port_instance** **)**

borrow_limit_spec ::= **(** *label*? **BORROW_LIMIT** NUMBER *port_instance** **)**

clock_delay_spec ::= **(** *label*? **CLOCK_DELAY** *clock_root leaf_spec+* **)**

clock_root ::= *port_instance*
||= **(** *cell_instance input_port output_port* **)**

leaf_spec ::= **(** *leaf_delay_spec+ port_instance** **)**

leaf_delay_spec ::= *insertion_delay_spec*
||= *clock_skew_spec*
||= *clock_slew_spec*

insertion_delay_spec ::= **(** **INSERTION_DELAY** *rise_fall_min_max* **)**

clock_skew_spec ::= **(** **SKEW** *min_max* **)**

clock_slew_spec ::= **(** **SLEW** *slew_value* **)**

waveform_edge_identifier ::= **POSEDGE**
||= **NEGEDGE**

edge_identifier ::= **POSEDGE**
||= **NEGEDGE**
||= **ANYEDGE**
||= **0z**
||= **z1**
||= **1z**
||= **z0**

Parasitics Subset

The parasitics subset is defined as follows:

parasitics_subset ::= **(SUBSET PARASITICS** *parasitics_subset_body* **)**

parasitics_subset_body ::= *parasitics_subset_spec+*
||= *include*

parasitics_subset_spec ::= *parasitics_environment*
||= *parasitics_constraints*
||= *extension*
||= *meta_data*

Parasitics Environment

The parasitics environment is defined as follows:

parasitics_environment ::= **(ENVIRONMENT** *parasitics_env_spec+* **)**

parasitics_env_spec ::= *parasitics_env_spec_0*
||= *parasitics_env_spec_1*

parasitics_env_spec_0 ::= *external_load_spec*
||= *extension*
||= *meta_data*

external_load_spec ::= **(** *label*? **EXTERNAL_LOAD** *capacitance port_instance** **)**

capacitance ::= *min_max*

parasitics_env_spec_1 ::= **(LEVEL** 1 *parasitics_env_1+* **)**

parasitics_env_1 ::= *parasitics_env_no_case_1*
||= *parasitics_env_case*

parasitics_env_no_case_1 ::= *external_fanout_spec*

external_fanout_spec ::= **(** *label*? **EXTERNAL_FANOUT** *num_loads port_instance** **)**

num_loads ::= *min_max*

parasitics_env_case ::= **(CASE** IDENTIFIER *parasitics_env_case_spec+* **)**

parasitics_env_case_spec ::= *parasitics_env_spec_0*
||= *parasitics_env_no_case_1*

Parasitics Constraints

The parasitics constraints are defined as follows:

parasitics_constraints ::= **(CONSTRAINTS** *parasitics_constraint+* **)**

parasitics_constraint ::= *parasitics_cnstr_spec_0*
||= *parasitics_cnstr_spec_1*

parasitics_cnstr_spec_0 ::= *internal_load_spec*
||= *load_spec*
||= *extension*
||= *meta_data*

internal_load_spec ::= **(** *label*? **INTERNAL_LOAD** *capacitance port_instance** **)**

load_spec ::= **(** *label*? **LOAD** *capacitance port_instance** **)**

parasitics_cnstr_spec_1 ::= **(** **LEVEL** 1 *parasitics_cnstr_1*+ **)**

parasitics_cnstr_1 ::= *parasitics_cnstr_no_case_1*
||= *parasitics_cnstr_case*

parasitics_cnstr_no_case_1 ::= *internal_fanout_spec*
||= *fanout_spec*

internal_fanout_spec ::= **(** *label*? **INTERNAL_FANOUT** *num_loadsport_instance** **)**

fanout_spec ::= **(** *label*? **FANOUT** *num_loads port_instance** **)**

parasitics_cnstr_case ::= **(** **CASE** IDENTIFIER *parasitics_cnstr_case_spec*+ **)**

parasitics_cnstr_case_spec ::= *parasitics_cnstr_spec_0*
||= *parasitics_cnstr_no_case_1*

Area Subset

The area subset is defined as follows:

area_subset ::= **(SUBSET AREA** *area_subset_body* **)**

area_subset_body ::= *area_cnstr_spec+*
||= *include*

area_cnstr_spec ::= *area_cnstr_spec_0*
||= *area_cnstr_spec_1*

area_cnstr_spec_0 ::= *primitive_area_spec*
||= *total_area_spec*
||= *extension*
||= *meta_data*

primitive_area_spec ::= **(** *label*? **PRIMITIVE_AREA** *area_value* **)**

total_area_spec ::= **(** *label*? **TOTAL_AREA** *area_value* **)**

area_value ::= *min_max*

area_cnstr_spec_1 ::= **(LEVEL** 1 *area_cnstr_1+* **)**

area_cnstr_1 ::= *area_cnstr_no_case_1*
||= *area_cnstr_case*

area_cnstr_no_case_1 ::= *porosity_spec*

porosity_spec ::= **(** *label*? **POROSITY** *porosity_value* **)**

porosity_value ::= *min_max*

area_cnstr_case ::= **(CASE** IDENTIFIER *area_cnstr_case_spec+* **)**

area_cnstr_case_spec ::= *area_cnstr_spec_0*
||= *area_cnstr_no_case_1*

Power Subset

The power subset is defined as follows:

power_subset ::= **(SUBSET POWER** *power_subset_body* **)**

power_subset_body ::= *power_cnstr_spec+*
||= *include*

power_cnstr_spec ::= *power_cnstr_spec_0*
||= *power_cnstr_spec_1*

power_cnstr_spec_0 ::= *average_cell_power*
||= *average_net_power*
||= *extension*
||= *meta_data*

average_cell_power ::= **(** *label*? **AVG_CELL_POWER** *power_value* **)**

average_net_power ::= **(** *label*? **AVG_NET_POWER** *power_value port_instance* **)**

power_value ::= *min_max*

power_cnstr_spec_1 ::= **(LEVEL** 1 *power_cnstr_1+* **)**

power_cnstr_1 ::= *power_cnstr_case*

power_cnstr_case ::= **(CASE** IDENTIFIER *power_cnstr_case_spec+* **)**

power_cnstr_case_spec ::= *power_cnstr_spec_0*

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Appendix 1

Cadence-Specific Extensions

CTLF_FILES

The locations of the Compiled Timing Library Format (CTLF) files which are to be used for a design are specified through GCF using an extension within the environment globals subset.

Syntax

env_globals_subset ::= **(GLOBALS_SUBSET ENVIRONMENT** *env_globals_body* **)**

env_globals_body ::= *env_globals_spec+*
||= *include*

env_globals_spec ::= *env_globals_spec_0*
||= *env_globals_spec_1*

env_globals_spec_0 ::= *process*
||= *voltage*
||= *temperature*
||= *operating_conditions*
||= *voltage_threshold*
||= *ctlf_files_extension*
||= *extension*
||= *meta_data*

ctlf_files_extension ::= **(EXTENSION** “CTLF_FILES” **(** *file_name+* **))**

file_name := IDENTIFIER

The file names can be relative or absolute path names. Relative path names are interpreted with respect to the directory in which the program which is reading the GCF is invoked.

Example

```
(GLOBALS_SUBSET ENVIRONMENT
  (EXTENSION “CTLF_FILES”
    (lib/mylib.ctlf
     lib/ram1.ctlf
     lib/ram2.ctlf
    )
  )
)
```