General Constraint Format Specification

Version 1.4

August 17, 1999

Cadence Design Systems, Inc.

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
555 River Oaks Parkway, MS 2B1
San Jose, CA 95134

Tel: (408) 428-5399
Fax: (408) 428-5959
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 1.4 - August 17, 1999

General Changes

- Introduced the notion of archaic constructs, which are supported in this version of GCF for backward compatibility, but may be dropped in the next major version. Classified a number of timing exceptions as archaic and grouped these into a separate section.
- Modified most constructs to allow the use of an asterisk as a place-holder for **NUMBER** or **RNUMBER** values that are unset.
- Expanded the Value Types section and consolidated definitions for the semantics of each value type. Added a description of the semantics for min/max value pairs, given a single operating point assumption.
- Clarified the Normal Precedence Rules to avoid conflicts with specific precedence rules for individual constructs.
- Added constructs to support explicitly specifying the types of ports, pins, instances, nets, and waveforms. In most cases, wherever an ambiguous name could be specified before, an explicitly typed name can be specified instead. Untyped names are still allowed for backward compatibility.
- Added constructs to support specifying ports, pins, instances, and nets using an expression including one or more asterisks as wildcards. In GCF 1.4, expressions are only allowed within the **DISABLE**, **MULTI_CYCLE**, **PATH_DELAY**, and **EXTERNAL_DELAY** constructs. This may be expanded to other constructs in a future version.

Signal Integrity Changes

- Added a **LIFETIME** construct to the environment globals subset, to model the required lifetime for the design.

Timing Analysis Changes

- Added a **SLEW_MODE** construct to select the algorithm used in merging slews.
- Enhanced the **WAVEFORM** construct to more precisely describe the ideal edges and jitter for the waveform.
- Enhanced the **DERIVED_WAVEFORM** construct to include **INVERT** and **PERIOD_DIVISOR** options.

Added an **EDGES** option to select particular edges from the master waveform and specify their phase shift.

Replaced the **SKEW_ADJUSTMENT** construct with **JITTER_ADJUSTMENT** for consistency with the **WAVEFORM** enhancements.

Added separate **PHASE_SHIFT** values for rise and fall.

Added an option to control whether the **PHASE_SHIFT** affects the ideal edge position or the effective edge position.

Revised the semantics of the **PERIOD_MULTIPLIER** option for improved compatibility.

Added an option to specify the duty cycle when using **PERIOD_DIVISOR**.

- Added a **CLOCK_ARRIVAL** construct to describe external insertion delay leading up to a *clock_root*.
- Added **REQUIRED** as a synonym for **DEPARTURE**, and modified the semantics to use required time as the preferred terminology over departure time.
- Modified the description of the **INTERNAL_SLEW** construct to use a *slew_value,* which is a *rise_fall_min_max* (four values), rather than just *rise_fall* (two values). There was an inconsistency in GCF 1.3 between the main description of the construct and the BNF summary; the BNF was correct.
- Added a **SLEW_LIMIT** construct as the preferred way to specify transition time constraints (both min and max). Included the ability to specify the **SLEW_LIMIT** on a master basis by giving the cell type and the port on that cell type.

  The **MAX_TRANSITION_TIME** construct is still supported but archaic.
- Greatly expanded the discussion of the various types of path specifications.
- Added an optional **BETWEEN** keyword to improve clarity in *endpoints_spec*.
- Added a *from_to_thru_spec* path specification to handle mixing from, to, and through options.
- Added precedence rules to handle cases where several timing exceptions affect the same path.
- Added a description of the semantics of the relationship between disables and slew and constant propagation.

- Added *cell_instance* and *waveform_name* as possible items for **BORROW_LIMIT**.
- Added a **DATA_LEAF** option to **CLOCK_DELAY** to handle cases where a clock signal is distributed to logic where it is treated as a data signal.
- Modified the semantics description of *rise_fall_min_max* values in **CLOCK_DELAY**.
- Added a **CLOCK_UNCERTAINTY** construct that specifies target-based and inter-clock skew (**CLOCK_DELAY** specifies intra-tree skew).
- Added a **CLOCK_MODE** construct to specify the default analysis mode (**IDEAL** or **ACTUAL**) for the clock networks within the design.
- Added a capability of overriding the default **CLOCK_MODE** on specific clock networks using the **CLOCK_DELAY** construct.

Parasitics Changes

- Added a **EXTERNAL_WIRE_LOAD_MODEL** construct that specifies the names of wire load models that are to be used for primary i/o ports.
- Added a **WIRE_LOAD_MODEL** construct that specifies the names of wire load models that are to be used for module instances and master cell types.
- Modified the **LOAD** limit construct to allow the limit to be specified on a master basis by giving the cell type and the port on that cell type.

Power Changes

- Modified the **TOTAL_AREA**, **PRIMITIVE_AREA**, **POROSITY, AVG_CELL_POWER**, and **AVG_NET_POWER** constraints to use the same convention as the other standard value types, where a single value represents both the min and max values (a range of a single point).
  - For the area and power constraints, a single value previously represented just the max endpoint of the range, and the min value was implicitly 0. The old semantics can be emulated by explicitly specifying 0 or by using an asterisk as a place-holder for the min value.
  - For the porosity constraint, a single value previously represented just the min endpoint of the range, and the max value was unspecified. The old semantics can be emulated by using an asterisk as a place-holder for the max value.

These changes were done for future consistency and are expected to have little impact because these constraints were not yet supported.

Version 1.3 - June 25, 1998

- Modified the semantics of the **MAX_TRANSITION_TIME** construct to allow the constraint to be specified on input ports as well as output and bidirectional ports.
- Modified the syntax and semantics of the **CLOCK_DELAY** construct to allow using a *waveform_name* to specifiy insertion delay and skew for external clock networks (virtual clocks).
- Modified the syntax of the **CLOCK_DELAY** construct to use a *rise_fall_min_max* value for **SKEW**.
- Modified the semantics of the **CLOCK_DELAY** construct to explicitly specify the interpretation of insertion delay and skew.
- Specified the semantics of delay calculation on interface nets given **DRIVER_CELL**, **DRIVER_STRENGTH**, **INPUT_SLEW**, and **EXTERNAL_LOAD** specifications, and the relationship between this and how **ARRIVAL**, **DEPARTURE**, and **EXTERNAL_DELAY** values should be set.
- Corrected typographical errors in, and clarified the semantics description for the **EXTERNAL_LOAD**, **INTERNAL_LOAD**, **EXTERNAL_FANOUT**, and **INTERNAL_FANOUT** constructs.
- Added an option to *disable_spec_0* to control whether paths through preset and clear inputs on registers are disabled, as well as an option to control whether reentrant bidirectional paths are disabled.
- Added an explicit statement in the precedence rules that default values propagate down through the hierarchy when specified on a non-leaf GCF cell, and the default can be overridden at lower levels in the hierarchy.
- Fixed several incorrect examples and added some diagrams.
- Clarified the definition of the starting points and ending points for endpoint-based false and multi-cycle exceptions, which are implicitly determined when waveform names or register names are specified as the **FROM** or **TO** item.
- Modified a number of places in the grammar where *meta_data*, a level 1 construct, was included as an option where only level 0 constructs should be used, to provide the same capability as option within the corresponding clauses where level 1 constructs should be used. This affects the way the grammar is organized, but doesn’t change the GCF syntax itself.

- Modified the syntax and semantics of the **WAVEFORM** construct to allow edge times to be negative numbers.

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 that 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 that 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

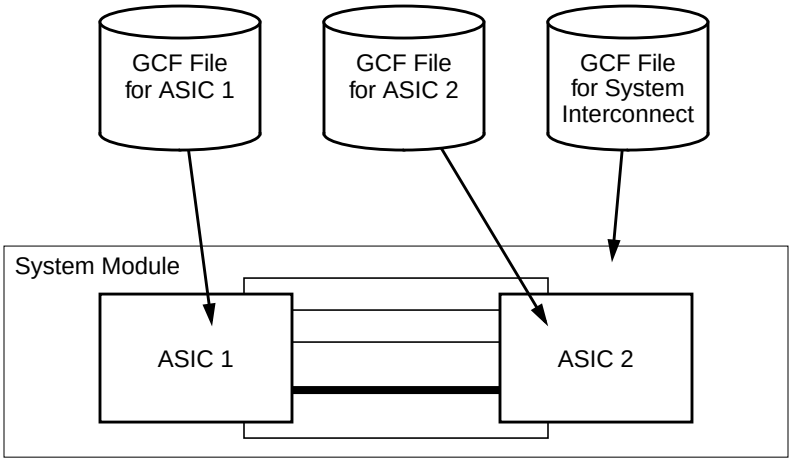
By accessing a GCF file, EDA tools are assured of consistent, accurate, and up-to-date data. This means that EDA tools can use data created by other tools as input to their own processes. By sharing data in this way, estimation, synthesis, floorplanning, analysis, and layout tools can all use a consistent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

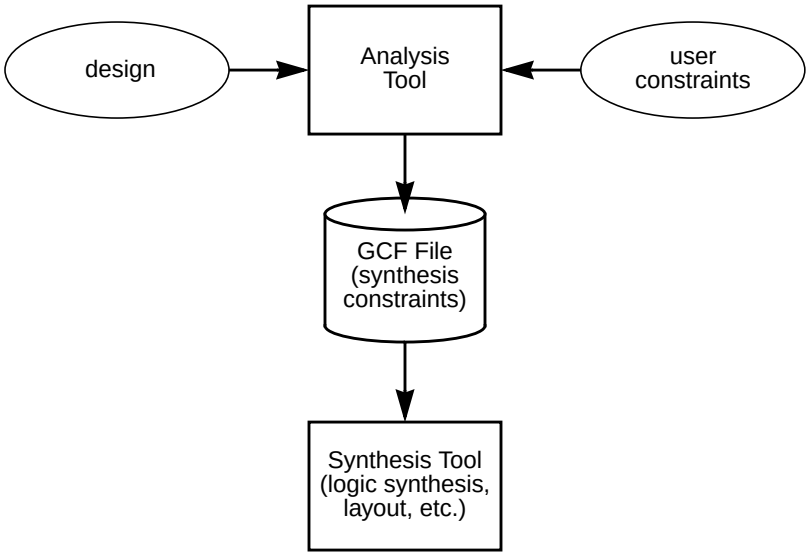
GCF includes a description of a standard semantics for many kinds of constraints. Some tools might not support all of the types of constraints in GCF, or might restrict the semantics for some types of constraints. For example, a layout tool might handle disabling of false paths where a single port is 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 32).

The *globals* section describes information that 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 77). 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 8, “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*
||= *current_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 1.4”)
```

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” “2.0” “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 193.

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* **)**
current_scale ::= **(CURRENT_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.

The default current scale is 1 ampere. If *current_scale* is specified, the GCF reader will multiply all current numbers in the GCF file by the specified value, which is in amperes. For example, a multiplier of 1.0E-3 corresponds to current values in milliamps.

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 that 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 that 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 44).

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 that 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 that 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 8, "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 that 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 8, “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 that contains the *port_instance* (see “Timing Environment” on page 84).

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 that 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.

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 that 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; see "Meta Data" on page 44.

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 that describes those constructs.

- A value that is given explicitly for a particular design element 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 design element, the value that appears later in the GCF file is used.
- If a place-holder ("*") is given for a value in one construct, and the same type of value is given explicitly in another construct of the same type, the explicit value is used.
- If a place-holder ("*") is given for a value in one construct, and the same type of value is given as a default in another construct of the same type, the default value is used.
- Default values affect the current GCF cell and all of its hierarchical descendents, unless overridden for a lower level cell.
- If two different default values are given at the same hierarchical level, the default value that appears later in the GCF file is used.

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.

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 that the tool supports; the remaining constraints in the set are ignored.

Other Meta Data

There are many other types of meta data that 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 that 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 47.

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.

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.

.

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 that 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 that 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.

.

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 that are legal. This section describes a number of basic value types that are used in other constructs.

The semantics for values that require both a minimum and a maximum value depends on the type of operating conditions that are specified for analysis. See “Min/Max Values and Operating Conditions” on page 51 for a description of the different interpretations that are possible.

Min and Max

Syntax

min_and_max ::= *min_number max_number*

r_min_and_max ::= *r_min_number r_max_number*

min_number ::= NUMBER

max_number ::= NUMBER

r_min_number ::= RNUMBER

r_max_number ::= RNUMBER

Two values must be specified for the *min_and_max* and *r_min_and_max* value types. The first represents the min value, while the second represents to the max value. Place-holders are not allowed for either value in the *min_and_max* and *r_min_and_max* value types.

NUMBER is a non-negative (zero or positive) real number, for example: 0, 1, 0.0, 3.4, .7, 0.3, 2., 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., 2.4e2, -5.3e-1, 8.2E+5

Min, Max, or both Min and Max

Syntax

```
min_max ::= NUMBER
        ||= min_value max_value
r_min_max ::= RNUMBER
          ||= r_min_value r_max_value
min_value ::= number_or_place_holder
max_value ::= number_or_place_holder
r_min_value ::= r_number_or_place_holder
r_max_value ::= r_number_or_place_holder
number_or_place_holder ::= NUMBER
                       ||= *
r_number_or_place_holder ::= RNUMBER
                         ||= *
```

One or two values can be specified for the *min_max* and *r_min_max* value types. When one value is specified, it applies to both the min and the max values. When two values are specified, the first represents the min value, while the second represents to the max.

Both value types allow an asterisk to be used as a place-holder for either the min or the max value. Whenever an asterisk is used as a place-holder, the corresponding value is treated as unspecified. Whenever several *number_or_place_holder* or *rnumber_or_place_holder* values appear in a row (as in *min_max*), at least one of the values must not be an asterisk.

Rise, Fall, or both Rise and Fall

Syntax

```
rise_fall ::= NUMBER
          ||= rise_value fall_value
r_rise_fall ::= RNUMBER
            ||= r_rise_value r_fall_value
rise_value ::= number_or_place_holder
fall_value ::= number_or_place_holder
r_rise_value ::= r_number_or_place_holder
r_fall_value ::= r_number_or_place_holder
```

The *rise_fall* and *r_rise_fall* value types represent a pair of times, one for a rise edge and one for a fall edge.

One or two values can be specified. If a single value is specified, it applies to both the rise and fall edges. If two values are specified, the first value represents the rise edge, and the second value represents the fall edge. When two values are specified, at least one of them must not be an asterisk.

Rise Min/Max, Fall Min/Max

Syntax

```
rise_fall_min_max ::= NUMBER
                  ||= rise_value fall_value
                  ||= rise_min_value rise_max_value
                        fall_min_value fall_max_value

r_rise_fall_min_max ::= RNUMBER
                    ||= r_rise_value r_fall_value
                    ||= r_rise_min_value r_rise_max_value
                          r_fall_min_value r_fall_max_value

rise_min_value ::= number_or_place_holder
rise_max_value ::= number_or_place_holder
fall_min_value ::= number_or_place_holder
fall_max_value ::= number_or_place_holder
r_rise_min_value ::= r_number_or_place_holder
r_rise_max_value ::= r_number_or_place_holder
r_fall_min_value ::= r_number_or_place_holder
r_fall_max_value ::= r_number_or_place_holder
```

The *rise_fall_min_max* and *r_rise_fall_min_max* value types represent a range of times for a rising edge and a range of times for a falling edge.

One, two, or four values can be specified. If a single value is specified, it applies to all four of the edge times.

If two values are specified, the first value applies to both the rise minimum and the rise maximum values, and the second value applies to both the fall minimum and the fall maximum values. In the two value forms, if an asterisk is used as a place-holder for the first value, the rise minimum and rise maximum values are unset. If an asterisk is used as a place-holder for the second value, the fall minimum and fall maximum values are unset. At least one of the two values must not be an asterisk.

When four values are specified, the order of the values is rise minimum, rise maximum, fall minimum, and fall maximum.

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

Min/Max Values and Operating Conditions

Most constraints and environment specifications in GCF allow a min/max pair of values (*r_min_max*), or min/max value pairs for rising and falling transitions (*r_min_max_rise_fall*).

In GCF 1.4, the **OPERATING_CONDITIONS** construct supports a single operating point. Therefore, the semantics for min/max values are defined as follows:

- A min/max value for a constraint represents the allowable variation of the constrained parameter, measured at the operating point
- A min/max value for an environment specification represents the extremes of the parameter expected at the operating point.

A future version of GCF is expected to support multiple operating points, in which case the semantics for min/max values will depend on whether a single operating point or multiple operating points are given.

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 an operating point 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*
||= *lifetime*
||= *extension*
||= *meta_data*

env_globals_spec_1 ::= **(LEVEL** 1 *env_globals_1*+ **)**
env_globals_1 ::= *env_globals_case*
||= *meta_data_1*
env_globals_case ::= **(CASE** IDENTIFIER *env_globals_case_spec*+ **)**
env_globals_case_spec ::= *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* that 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* that 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* that 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 that 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 that 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 that 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 that 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 that were used in calculating the slews (transition times) in the GCF file.

The measurement points are defined as a percentage 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 for slew measurement 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 for slews as numbers between 0 and 100.

Example

```
(voltage_threshold 20.0 80.0)
```

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

Lifetime

The *lifetime* construct specifies the required operating lifetime for the design, which is used in some types of signal integrity and reliability analysis.

Syntax

lifetime ::= **(** *label?* **LIFETIME** *lifetime_value* **)**

lifetime_value ::= *min_max*

The *lifetime_value* specifies the required lifetime in years. Although this is a time value, it is not scaled by the *time_scale*, which is usually intended to scale time values to be in units of ns or ps.

In GCF 1.4, only a single operating point can be modeled with the **OPERATING_CONDITIONS** construct. This leads to ambiguities because *lifetime_value* supports both minimum and maximum fields, for compatibility with a future version of GCF that is expected to support multiple operating points. At that time, the minimum fields will correspond to best case operating conditions while the maximum fields will correspond to worst case operating conditions.

For GCF 1.4, in general the mininimum and maximum fields in *lifetime_value* should both be set to the same value:

- The minimum required lifetime expected at the operating point specified in the **OPERATING_CONDITIONS** construct.

Tools will generally use the minimum field.

Example

```
(lifetime 3)
```

In this example, the design must operate successfully for at three years at the given operating point.

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 that 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 ::= *slew_mode*
||= *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*
||= *meta_data_1*

The following sections describe how operating points are specified for use in delay calculation and timing analysis, primary waveforms, derived waveforms, clock groups, and case-dependent timing globals.

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.

Slew Mode

The **SLEW_MODE** construct specifies how slews should be propagated through the design.

Syntax

slew_mode ::= **(** *label?* **SLEW_MODE** *slew_mode_value* **)**

slew_mode_value ::= **WORST**
||= **CRITICAL**

The default is **WORST**.

Example

```
(SLEW_MODE CRITICAL)
```

When the **SLEW_MODE** is set to **WORST**,

- The smallest of the minimum incoming slews for each timing arc will be used in computing the minimum delays for SDF and the earliest clock and data arrival times for timing checks.
- The largest of the maximum incoming slews for each timing arc will be used in computing the maximum delays for SDF and the latest clock and data arrival times for timing checks.
- The average of the typical incoming slews for each timing arc will be used in calculating the typical delays for SDF.

When the **SLEW_MODE** is set to **CRITICAL**,

- The minimum slew of the earliest transition arriving at the start of each timing arc will be used in computing the minimum delays for SDF and the earliest clock and data arrival times for timing checks.
- The maximum slew of the latest transition arriving at the start of each timing arc will be used in computing the maximum delays for SDF and the latest clock and data arrival times for timing checks.
- The average of the typical incoming slews for each timing arc will be used in calculating the typical delays for SDF.

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** *edge_position* **)**

neg_edge ::= **(** **NEGEDGE** *edge_position* **)**

edge_position ::= *ideal_edge*
||= *ideal_edge_with_jitter*
||= *edge_range*

ideal_edge ::= RNUMBER
||= *placeholder*

ideal_edge_with_jitter ::= *ideal_edge jitter_spec*

jitter_spec ::= **(** **JITTER** *jitter_value* **)**

jitter_value ::= NUMBER
||= *neg_jitter pos_jitter*

neg_jitter ::= NUMBER
pos_jitter ::= NUMBER

edge_range ::= *r_min_and_max* (archaic)

The name of the waveform must be unique. The period describes the interval at which the waveform repeats, and it 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 67), there is one implicit reference point for each clock domain that 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 edge times may be negative, in which case care must be taken to correctly define the period, which is always a positive number.

The offset of each edge can be specified in three different ways. The simplest form is to specify only the ideal offset. Given an ideal clock waveform that has no uncertainty, the rising and falling edges will modeled as always occuring at exactly the same offset within every clock cycle.

Placeholders must only be used in the **DERIVED_WAVEFORM** construct for an *ideal_edge*. Placeholders must not be used for an *ideal_edge* in a **WAVEFORM** construct.

Modeling Jitter

External factors in the environment such as crosstalk can introduce variation called jitter in the actual offset for a clock edge from cycle to cycle. The second form for the *edge_position* should be used to model the peak jitter that may occur, where peak jitter is the maximum difference in any cycle between the actual offset for an edge and the ideal position for the edge.

The *neg_jitter* value is subtracted from the *ideal_edge* and the *pos_jitter* value is added to the *ideal_edge* to create an uncertainty region, where the actual edge position in a particular cycle may lie anywhere within the uncertainty region. For any *edge_position* that has an uncertainty region, tools will assume that a single transition of the specified direction occurs somewhere in the uncertainty region but will not make any assumptions about the exact location. Tools unable to model uncertainty will issue a warning message and use the *ideal_edge* position instead.

When the edge positions of two waveforms are compared in order to establish the relationship between the waveforms, the *ideal_edge* is always used.

When a waveform edge is used as a reference for an arrival or departure time, jitter on the waveform edge extends the uncertainty region for the arrival or departure time.

The *r_min_and_max* form of the *edge_position* is archaic and has been replaced by the *ideal_edge_with_jitter* form. The uncertainty region is treated the same with both forms, but in the *r_min_and_max* form the *ideal_edge* is defined as the mean of the two endpoints, rather than represented explicitly. Computing the mean is subject to floating point arithmetic inaccuracies that can affect the comparison between *ideal_edges*. Two waveforms intended to have the same *ideal_edge* and

uncertainty regions with different sizes may not have exactly the same computed *ideal_edge* position.

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).

Example

```
(WAVEFORM “100 MHz 50/50” 10.0
  (POSEDGE 0 (JITTER 0.2))
  (NEGEDGE 5.0 (JITTER 0.2))
)
```

In this example, a waveform is defined with a jitter of 0.2 ns on both the rising and falling edges. This creates an uncertainty window of 0.4 ns around each edge. The earliest possible transition for the rising edge in any machine cycle is expected to be at an offset of -0.2 ns from the implicit reference point. The latest possible transition for the falling edge in any machine cycle is expected to be at an offset of 5.2 ns from the implicit reference point.

Example

```
(WAVEFORM “100 MHz 50/50” 10.0
  (POSEDGE 0 (JITTER 0.1 0.2))
  (NEGEDGE 5.0 (JITTER 0.3 0.4))
)
```

In this example, a waveform is defined to have different positive and negative jitter values for each edge. The earliest possible transition for the rising edge is expected to be at an offset of -0.1 ns, while the latest possible transition is at 0.2 ns. The earliest possible transition for the falling edge is expected to be at an offset of 4.7 ns, while the latest possible transition is at 5.4 ns.

Example (archaic)

```
(WAVEFORM “100 MHz 50/50” 10.0
  (POSEDGE -0.2 0.2) (NEGEDGE 4.8 5.2)
)
```

This example illustrates using the archaic uncertainty range form to describe jitter of 0.2 ns. The *ideal_edge* position is determined as the mean

of the two endpoints for each edge, so the rising *ideal_edge* will be at approximately 0.0 ns, and the falling *ideal_edge* will be at approximately 5.0 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
derived_waveform_option+ **)**

parent_waveform_name ::= QSTRING

derived_waveform_option ::= *period_multiplier*
||= *period_divisor*
||= *derived_edges*
||= *phase_shift*
||= *jitter_adjustment*
||= *invert*

period_multiplier ::= **(PERIOD_MULTIPLIER**
period_multiplier_value **)**

period_divisor ::= **(PERIOD_DIVISOR**
period_divisor_value duty_cycle_value? **)**

derived_edges ::= **(EDGES** *derived_edge_list* **)**

derived_edge_list ::= *derived_pos_pair+*
||= *derived_neg_pair+*

derived_pos_pair ::= *derived_pos_edge derived_neg_edge*

derived_neg_pair ::= *derived_neg_edge derived_pos_edge*

derived_pos_edge ::= **(POSEDGE** *derived_edge* **)**

derived_neg_edge ::= **(NEGEDGE** *derived_edge* **)**

derived_edge ::= *edge_num derived_edge_shift?*

derived_edge_shift ::= **(PHASE_SHIFT** *edge_shift_value* **IDEAL***?* **)**

phase_shift ::= **(PHASE_SHIFT** *phase_shift_value* **IDEAL***?* **)**

jitter_adjustment ::= **(JITTER_ADJUSTMENT** *edge_pair_list* **)**

invert ::= **INVERT**

period_multiplier_value ::= DNUMBER

period_divisor_value ::= DNUMBER

duty_cycle_value ::= NUMBER

edge_num ::= DNUMBER

edge_shift_value ::= RNUMBER

phase_shift_value ::= *r_rise_fall*

The basic relationshipship between the edges in the parent waveform and the edges in the derived waveform can be specified in two different ways:

- using the *period_multiplier* and/or the *period_divisor* constructs to scale the period and edge positions of the parent waveform
- using the *derived_edges* construct to select specific edges by number from multiple cycles of the parent waveform

These two approaches cannot be combined. The *derived_edges* construct cannot be used in combination with *period_multiplier* or *period_divisor*.

Uniform Scaling

If a *period_multiplier* is specified, the period of the derived waveform is obtained by multiplying the period of the parent waveform by the *period_multiplier_value.*

- If the *period_multipler* is a power of two, the positions of each of the edges in the derived waveform will be set to the position of successive rising edges across multiple periods of the parent waveform, starting with the first rising edge in the first period.
- If the *period_multiplier* is not a power of two, the position of each waveform edge in the parent is multiplied by *period_multiplier* to determine the corresponding edge position in the derived waveform.

If a *period_divisor* is specified, the period of the derived waveform is obtained by divided the period of the parent waveform by the *period_divisor_value.* The *duty_cycle_value* represents the percentage (0 to 100) of the derived period that the derived waveform is high. If *duty_cycle_value* is not specified, the position of each waveform edge in the parent is also divided, to determine the corresponding edge position in the derived waveform.

If *duty_cycle_value* is specified, only the first edge position in the parent is used in determining the edge positions in the derived waveform, which

will always have just two edges, regardless of how many edges there are in the parent waveform. The position of the first waveform edge in the parent is divided by the *period_divisor_value* to obtain the position of the first derived waveform edge. Then the *duty_cycle_value* is applied to the derived period to determine the second edge position.

- If the first waveform edge in the parent is a rising edge, or if it is falling and the **INVERT** keyword is specified, then the second edge position will be at time
  - first_edge_time + ((*duty_cycle_value* / 100.0) * derived_period)
- If the first waveform edge in the parent is a falling edge, or if it is rising and the **INVERT** keyword is specified, then the second edge position will be at time
  - first_edge_time + (((100 - *duty_cycle_value*) / 100.0) * derived_period)

Both a *period_multiplier* and a *period_divisor* can be specified, for cases where the period of the derived waveform is a rational multiple of the parent waveform's period. The multiplier is applied first, then the divisor.

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)
  )
)
```

In this example, which models a clock divider, the period of the derived waveform is multiplied by 2 (and the frequency is divided by 2). The rising edge of the derived waveform is at 0, and the falling edge of the derived waveform is at 10.

Edge Selection

If *derived_edges* is specified, the edge positions in the derived waveform are obtained by selecting particular edges by number from multiple cycles of the parent waveform. Each derived edge can then be shifted by a unique amount using *derived_edge_shift*.

By default, the ideal edge position is the same as the ideal edge position in the parent, and the *derived_edge_shift* is treated as insertion delay. If the **IDEAL** keyword is specified in the *derived_edge_shift* construct, the *derived_edge_shift* is included in the ideal edge position.

The edge numbers in the parent waveform are consecutive integers starting at 1 and incrementing on each transition.

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”
    (edges (posedge 1) (negedge 3))
  )
)
```

For example, for a parent waveform with edges at 0 (rising) and 5 (falling) and a period of 10, specifying edge numbers 1 and 3 in *derived_pos_pair* will result in a waveform with edges at 0 (rising) and 10 (falling).

Uniform Phase Shift

If *phase_shift* is specified, all of the edges of the derived waveform are computed by adding the specified value(s) to the corresponding edge positions specified in the parent waveform or to the computed edge positions if *period_multiplier*, *period_divisor*, or *derived_edges* is specified.

- If the **INVERT** keyword is not specified, the rise *phase_shift_value* is added to the *pos_edge* edges, while the fall *phase_shift_value* is added to the *neg_edge* edges.
- If the **INVERT** keyword is specified, the rise *phase_shift_value* is added to the *neg_edge* edges, while the fall *phase_shift_value* is added to the *pos_edge* edges.
- If both *phase_shift* and *derived_edge_shift* are specified, the sum of the two is used in computing the *derived_edge* position.

Jitter Adjustments

If *jitter_adjustment* is not specified, the derived waveform will have the same jitter as the parent waveform.

If *jitter_adjustment* is specified, it overrides the jitter from the parent waveform. Within the *jitter_adustment* construct, a placeholder must be used to represent each *ideal_edge* position, since the actual offset of each *ideal_edge* will be computed from the other specifications.

When a combination of *period_multiplier*, *period_divisor*, *derived_edges*, *phase_shift*, or *jitter_adjustment* constructs are specified, first the ideal edge positions for the derived waveform are computed, using the *period_multiplier* or *period_divisor* if specified.

Then the effective edge positions are computed, considering the effect of a *phase_shift* if specified. Finally, the uncertainty around each effective edge position is determined from the *jitter_adjustment* if specified

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.

If the **INVERT** option is specified, the derived waveform is inverted with respect to its parent.

When the edge positions of a derived waveform are compared against another waveform in order to establish the relationship between the waveforms, the comparison is done using the ideal edges for the derived waveform.

When the **MULTI_CYCLE** construct (see “Multi-Cycle Paths” on page 129) 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 both Level 0 and Level 1, by default all clock waveforms are assigned to the same default clock domain. In Level 1, it is possible to describe cases where not all of the clocks are derived from the same source by separating the waveforms into groups of related clocks or “clock domains.” If any clock domains are specified, only paths between clock waveforms in the same group are constrained.

Clock waveforms in different domains are assumed to be asynchronous. There is no default constraint on the delay of paths that 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, 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 that 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.

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

Cell Instance

The cell instance construct is used to identify a particular instance of a cell within the design. In early versions of GCF, the cell instance construct was untyped. In some constructs, it was ambiguous without access to the netlist whether a given string or *prefix_id* represented an cell instance name or a port name or a pin name. The *typed_cell_instance* construct can be used to avoid this ambiguity, but for backward compatibility it is not required.

Syntax

cell_instance ::= *untyped_cell_instance*
||= *typed_instance_list*

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

typed_instance_list ::= **(INSTANCE** *untyped_cell_instance*+ **)**

Port Instance

The port instance construct is used to identify either a top level port on the current GCF cell, a pin on a primitive contained within the current GCF cell or its descendents, or a pin on a hierarchical module contained within the current GCF cell or its descendents (a “hierarchical pin”).

Not all tools reading GCF support hierarchical pins, because doing so requires access to a hierarchical netlist, while some tools only read a flattened netlist. Constraints originally specified on hierarchical pins may need to be “flattened”, or propagated to primitive pins at certain points in a design flow.

In early versions of GCF, the port instance construct was untyped. In some constructs, it was ambiguous without access to the netlist whether a given string or *prefix_id* represented an cell instance name or a port name or a pin name. The *typed_port_instance* construct can be used to avoid this ambiguity, but for backward compatibility it is not required.

Syntax

port_instance ::= *untyped_port_instance*
||= *typed_port_instance*

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

typed_port_instance ::= *typed_port_list*
||= *typed_pin_list*

typed_port_list ::= **(PORT** *untyped_port_instance*+ **)**

typed_pin_list ::= **(PIN** *untyped_port_instance*+ **)**

Example

```
(CELL()
  (SUBSET “timing”
    (EXCEPTIONS
      (SLEW_LIMIT 1.0 2.0 3.0 4.0 a.b.c.d.IN1)
      (SLEW_LIMIT 5.0 6.0 7.0 8.0 a.b.c.e.IN1)
    )
  )
)
```

In this example, a slew limit (transition time) constraint on two primitive pins is specified using the *untyped_port_instance* form.

Example

```
(CELL()
  (SUBSET “timing”
    (EXCEPTIONS
      (SLEW_LIMIT 1.0 2.0 3.0 4.0 (PIN a.b.c.d.IN1))
      (SLEW_LIMIT 5.0 6.0 7.0 8.0 (PIN a.b.c.e.IN1))
    )
  )
)
```

In this example, the same constraint is specified using the *typed_pin_list* form.

Example

```
(CELL()
  (NAME_PREFIXES 2
    0 “a.b.c.d.”
    1 “a.b.c.e.”
  )
  (SUBSET “timing”
    (EXCEPTIONS
      (SLEW_LIMIT 1.0 2.0 3.0 4.0 (1 IN1))
      (SLEW_LIMIT 5.0 6.0 7.0 8.0 (2 IN1))
    )
  )
)
```

In this example, the same constraint is specified using the name prefixes form.

Net

The net construct is used to identify a particular net contained within the current GCF cell or its descendents. In most GCF constructs, nets are identified implicitly by specifying one of the pins connected to the net. However, in several constructs a net name can be used directly.

Generally, the name of a net that connects through several levels of hierarchy is ambiguous, as the net will have a different local name, or alias,

within each level of hierarchy. Applications that interpret net names generally need to have access to all of the net aliases in order to find a net referenced in the GCF.

The net name can be typed or untyped, for consistency with cell instance and port instance. The typed form is preferred.

Syntax

net ::= *untyped_net*
||= *typed_net_list*

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

typed_net_list ::= **(NET** *untyped_net+* **)**

Example

```
(CELL()
  (SUBSET “timing”
    (EXCEPTIONS
      (DISABLE
        (PATHS
          (THRU_ALL
            (NET net1)
            (NET net2)
          )
        )
      )
    )
  )
)
```

In this example, all paths through both *net1* and *net2* are disabled.

Typed Waveform

For consistency with *cell_instance* and *port_instance*, there is an explicitly typed form for waveforms. The *typed_waveform_list* can be used for clarity, although there isn’t any ambiguity between waveforms and other design objects, since waveform names must be enclosed in quotes.

Syntax

typed_waveform_list ::= **(WAVEFORM** *waveform_name+* **)**

Example

```
(WAVEFORM “wave1”)
```

Instance, Port, Pin, and Net Expressions

In certain GCF constraints, to reduce the GCF file size it is possible to specify an expression including the WILDCARD character, “*”, that matches a set of cell instance names, port names, or pin names.

Syntax

typed_instance_expr ::= **(INSTANCE_EXPR** PATH_EXPR **)**

typed_port_expr ::= **(PORT_EXPR** PATH_EXPR **)**

typed_pin_expr ::= **(PIN_EXPR** PATH_EXPR **)**

typed_net_expr ::= **(NET_EXPR** PATH_EXPR **)**

PATH_EXPR is the same as PATH (see “Variables” on page 193), with the addition of the wildcard character. The WILDCARD character matches any substring within a single level of a hierarchical name, but it does not match across hierarchy boundaries.

Example

```
(CELL()
  (SUBSET “timing”
    (EXCEPTIONS
      (DISABLE (PIN_EXPR a.*.c.*.IN1))
    )
  )
)
```

In this example, given that the current GCF cell contains the pins a.b.c.d.in1, a.b.c.e.in1, and a.b.c.e.f.in1, only first two pins will be disabled. The third pin would be matched by

```
(PIN_EXPR a.*.c.*.*.in1)
```

Example

```
(CELL()
  (SUBSET “timing”
    (EXCEPTIONS
      (DISABLE (PIN_EXPR *.IN1))
    )
  )
)
```

In this example, only the IN1 pins on instances that are direct children within the current GCF cell are matched, not all of the IN1 pins on any instance contained within the current GCF cell and its descendents.

Example

```
(CELL()
  (SUBSET “timing”
    (EXCEPTIONS
      (DISABLE (FROM (PORT_EXPR SCAN_DATA_*_)))
    )
  )
)
```

In this example, a SCAN_DATA bus port in the original netlist has been mapped into individual ports in the current netlist, replacing the bus delimiters with underscores. The **PORT_EXPR** construct is used to match all of the ports corresponding to the original bus port.

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 that contains the cell. The view name specifies a particular view of the cell.

Example

```
(CELLTYPE "AN2")
```

This example specifies the AN2 cell type. Since the library is not specified, the effect of this ambiguous if there are several libraries used in the design that include different cells named AN2.

Example

```
(CELLTYPE "REFLIB" "AN2")
```

This example specifies the AN2 cell from the REFLIB library.

Port Master

The *port_master* symbol is used to refer to a port on a particular type of cell. This is generally used to establish a master-based default for a constraint on all *port_instances* that correspond to the *port_master*.

Syntax

port_master ::= **(** *cell_id scalar_port* **)**

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

Example

```
((CELLTYPE "REFLIB" "AN2") IN1)
```

This example specifies the IN1 port on the AN2 cell from the REFLIB library.

Port Instance or Master

The *port_instance_or_master* symbol is used to refer to either a specific *port_instance*, or to all *port_instances* that correspond to a *port_master*.

Syntax

port_instance_or_master ::= *port_instance*
||= *port_master*

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 25). 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 that 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 that 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 tools to efficiently access only the data that is relevant to them.

Syntax

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

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")
  (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 required 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*
||= *clock_arrival_spec*
||= *arrival_spec*
||= *required_spec*
||= *external_delay_spec*
||= *driver_spec*
||= *input_slew_spec*
||= *extension*

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*
||= *meta_data_1*

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*
clock_root+ **)**

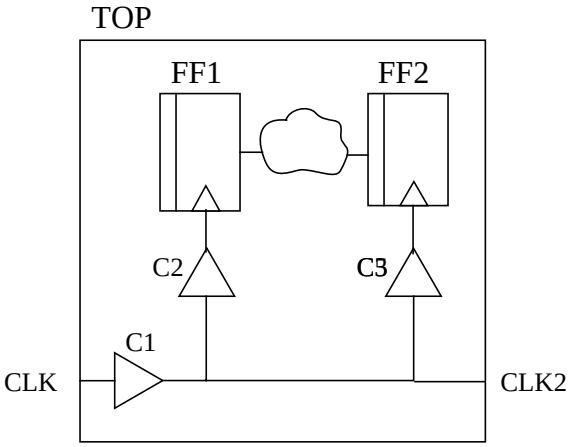
clock_root ::= *port_instance*

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 *clock_root* can be a primary input port, an output of a primitive instance within the current GCF cell, or a hierarchical output pin on a lower level cell.

An error message will be given if a *clock_root* for a **CLOCK** construct lies in the transitive fanout of a *clock_root* for another **CLOCK** construct. When modeling hierarchical clock trees, each GCF must only specify the highest *clock_root* contained within the portion of the design described by the GCF.

Figure 3 Simple Clock Tree



Example

```
(CLOCK “100 MHz 50/50” CLK)
```

In Figure 3, the *CLK* input port is a *clock_root* that is the source of the “100 MHz 50/50” waveform within the *TOP* module. The clock network

distributes that waveform to the clock input pins of *FF1* and *FF2*, as well as to the output port, *CLK2*.

Clock Arrival

The **CLOCK_ARRIVAL** construct specifies external insertion delay that should be included in the effective offset for certain clock edges.

Syntax

clock_arrival_spec ::= **(** *label?* **CLOCK_ARRIVAL**
clock_arrival_value
clock_arrival_item+ **)**

clock_arrival_value ::= *r_rise_fall_min_max*

clock_arrival_item ::= *clock_root*
||= *clock_leaf*
||= *waveform_name*
||= *typed_waveform_list*

clock_leaf ::= *port_instance*

The *clock_arrival_value* is a time value and must be specified in the units defined by the *time_scale*. It follows the ordering convention for *r_rise_fall_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

When the current GCF cell is part of a larger design, the current GCF cell may contain only a portion of a larger clock distribution network. The characteristics of the overall clock network are important when the current GCF cell contains two different subtrees. In that case, the relationship between clock edges at registers in different subtrees depends on the external insertion delay from the root of the overall clock network up to each subtree’s *clock_root*. This external insertion delay should be specified using the **CLOCK_ARRIVAL** construct.

Generally, the effective offset of a clock edge at a register clock input includes:

- the offset of the clock edge within the waveform
- external insertion delay specified in the **CLOCK_ARRIVAL**
- internal insertion delay between the *clock_root* and the register clock input.

Similar factors also affect clock edges that are used as a reference for **ARRIVAL** and **REQUIRED** times. In that case, the whole clock network lies outside the current GCF cell, and the effective offset of the reference clock edge includes:

- the offset of the clock edge within the waveform
- a portion of the overall external insertion delay specified using a waveform name in the **CLOCK_ARRIVAL** construct
- a portion of the overall external insertion delay specified using a waveform name in the **CLOCK_DELAY** construct

Separating the overall external insertion delay into two parts can be valuable when the same clock waveform is used for both internal and external registers.

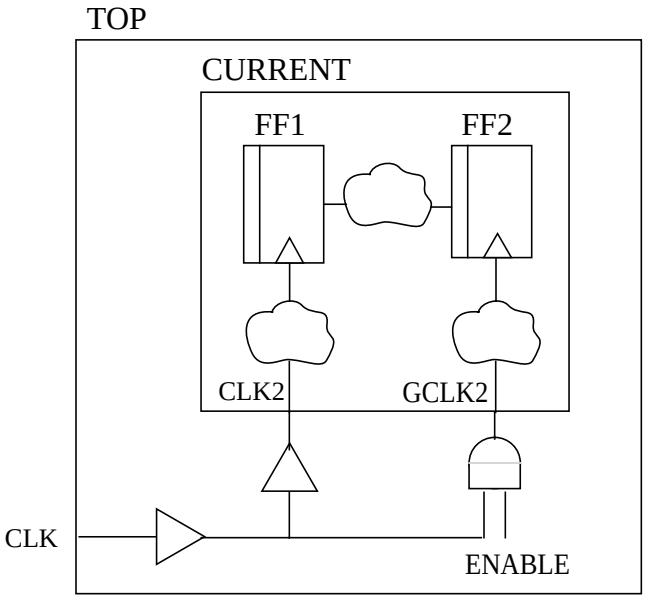
When a higher level subtree in the clock distribution network is not yet implemented, there may be uncertainty in the clock arrival time. Since the min/max range for the *clock_arrival_value* is used to model different operating points, it must not be used to describe the uncertainty in the higher level subtree. Instead, the uncertainty should be modeled using the **CLOCK_SKEW** construct (see “Inter-Clock Uncertainty” on page 151).

For a *clock_root*, the *clock_arrival_value* affects the effective offset of clock edges at the registers in the transitive fanout from the clock_root.

For a *clock_leaf*, the *clock_arrival_value* affects the effective offset of the clock edge at that *port_instance*, which must be the clock input of a register.

For a *waveform_name*, the *clock_arrival_value* affects the effective offset of the clock edges at the registers in the transitive fanout from each *clock_root* associated with the waveform. It also affects the effective offset of the clock edges used as a reference in **ARRIVAL** and **REQUIRED** constructs.

Figure 4 Hierarchical Clock Tree



Example

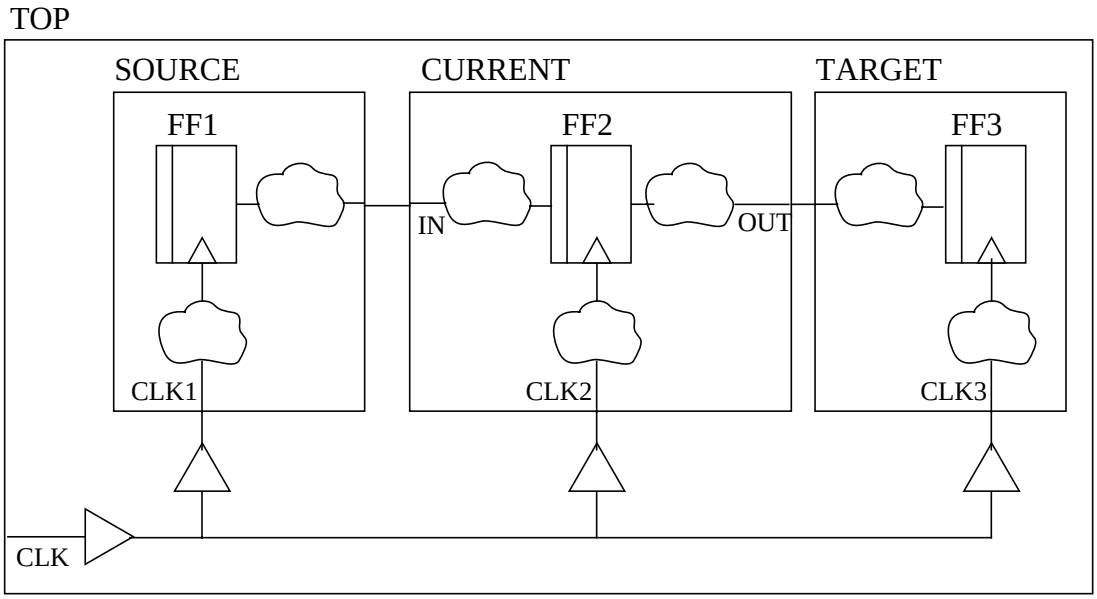
```
(CELL ()
  (SUBSET TIMING
    (CLOCK "WAVE" CLK2 GCLK2)
    (CLOCK_ARRIVAL 0.5 0.6 0.4 0.5 CLK2)
    (CLOCK_ARRIVAL 0.6 0.7 0.5 0.6 GCLK2)
  )
)
```

In Figure 4, the *CLK* input port on the *TOP* level module is the *clock_root* when the entire design is being analyzed.

However, for a GCF that is intended to describe the *CURRENT* module, *CLK2* and *GCLK2* are the *clock_roots*. Since they are derived from a common waveform, they can be listed in the same **CLOCK** construct. However, the partial insertion delay from the *CLK* input port at the *TOP* level to the *CLK2* input is different than the partial insertion delay from the *CLK* input port to the *GCLK2* input.

This difference in the top level insertion delays affects the effective constraint on paths between *FF1* and *FF2*. In this example, the clock arrival time at *GCLK2* is 100 ps later than at *CLK2*. This causes the effective setup constraint on paths between *FF1* and *FF2* to be 100 ps looser than if the clock arrival times were the same.

Figure 5 External Clock Trees and Arrival/Required Times



Example

```
(GLOBALS
  (GLOBALS_SUBSET TIMING
    (WAVEFORM "WAVE" 10 (NEGEDGE 0) (POSEDGE 5))
  )
)
(CELL ()
  (SUBSET TIMING
    (ENVIRONMENT
      (CLOCK "WAVE" CLK2)
      (CLOCK_ARRIVAL 0.5 0.7 0.4 0.6 "WAVE")
      (ARRIVAL (POSEDGE "WAVE") 5.0 6.5 4.0 5.2 IN)
      (REQUIRED (POSEDGE "WAVE") 4.0 2.7 3.0 2.0 OUT)
    )
    (EXCEPTIONS
      (CLOCK_DELAY "WAVE" (INSERTION_DELAY 2.0 2.6))
    )
  )
)
```

In a GCF for the *CURRENT* module in Figure 5, *CLK2* is the *clock_root* for the internal clock network. The top level clock tree affects the arrival time of the clock edge at the CLK2 input. The top level clock tree also affects the arrival time of the clock edge at CLK1 and CLK3, which in turn affects the arrival time at *IN* and the required time at *OUT*.

Specifying the waveform name in the **CLOCK_ARRIVAL** construct is a convenient way to describe a balanced top level clock network, where the partial insertion delay is the same from *CLK* to *CLK1*, *CLK2*, and *CLK3*.

Specifying the waveform name in the **CLOCK_DELAY** construct is a convenient way to describe balanced lower level clock networks, where the partial insertion delay within each of the lower level modules is the same.

The effective minimum offset of the rising clock edge at the clock input of FF2 is

5.0 (waveform edge offset) +
0.5 (external insertion delay from **CLOCK_ARRIVAL**) +
2.0 (internal insertion delay from **CLOCK_DELAY**)
= 7.5

The effective maximum offset of the rising clock edge at the clock input of FF2 is

5.0 (waveform edge offset) +
0.7 (external insertion delay from **CLOCK_ARRIVAL**) +
2.6 (internal insertion delay from **CLOCK_DELAY**).

The effective offsets of the reference clock edges used in the **ARRIVAL** and **REQUIRED** time constructs are computed similarly, except that the **CLOCK_ARRIVAL** construct describes the top level portion of the external clock network, and the **CLOCK_DELAY** construct describes the lower level portion of the external clock network.

The earliest arrival time of the falling edge at *IN* has a total offset from the implicit reference point of

5.0 (waveform edge offset) +
0.5 (partial external insertion delay from **CLOCK_ARRIVAL**) +
2.0 (partial external insertion delay from **CLOCK_DELAY**) +
4.0 (arrival time)
= 11.5

The latest time (as an offset from the implicit reference point) by which the rising data edge must reach *OUT* for single-cycle operation is

10.0 (cycle time) +
5.0 (waveform edge offset) +
0.7 (partial external insertion delay from **CLOCK_ARRIVAL**) +
2.6 (partial external insertion delay from **CLOCK_DELAY**) -
4.0 (required setup time)
= 14.3

The earliest time (as an ofset from the implicit reference point) that the falling data edge must not reach *OUT* before is

5.0 (waveform edge offset) +
0.5 (partial external insertion delay from **CLOCK_ARRIVAL**) +
2.0 (partial external insertion delay from **CLOCK_DELAY**) +
2.0 (required hold time)
= 9.5

Arrival Time

The **ARRIVAL** construct defines ranges of time in which signal transitions can occur at a *port_instance* that includes register data inpus 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 pins on primitives, and for hierarchical pins on lower level modules or blocks. When specified on internal or hierarchical pins, the arrival time overrides any propagated arrival time.

Syntax

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

arrival_waveform_edge ::= **(** *waveform_edge_identifier waveform_name* **)**

arrival_value ::= *source_arrival_value*

source_arrival_value ::= *r_rise_fall_min_max*
||= (waveform_edge_identifier
r_min_max) (archaic)

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

The *arrival_waveform_edge* specification, which identifies a waveform and an edge of that waveform, is required. The effective offset of the waveform edge implicitly includes any external insertion delay specified for the waveform using the **CLOCK_ARRIVAL** construct, as well as the offset of the edge specified in the waveform definition.

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 have only a single pair of edges.

The *arrival_value* is interpreted as a positive offset from the effective position of the waveform edge, and it affects all partial paths starting at the specified *port_instances*.

The *arrival_value* is a time value and must be specified in the units defined by the *time_scale*. It follows the ordering conventions for *r_rise_fall_min_max* and *r_min_max* described in “Value Types” on page 48, as well as the semantics described in “Min/Max Values and Operating Conditions” on page 51.

The *r_rise_fall_min_max* value type is the preferred form for the *source_arrival_value*.

The second *source_arrival_value* form, waveform_edge_identifier r_min_max, is archaic. It is more easily and consistently specified using the *r_rise_fall_min_max* form with asterisks as place-holders.

The arrival time at an input pin should include

- The portion of the external insertion delay of the clock network to the source register specified using the **CLOCK_ARRIVAL** construct

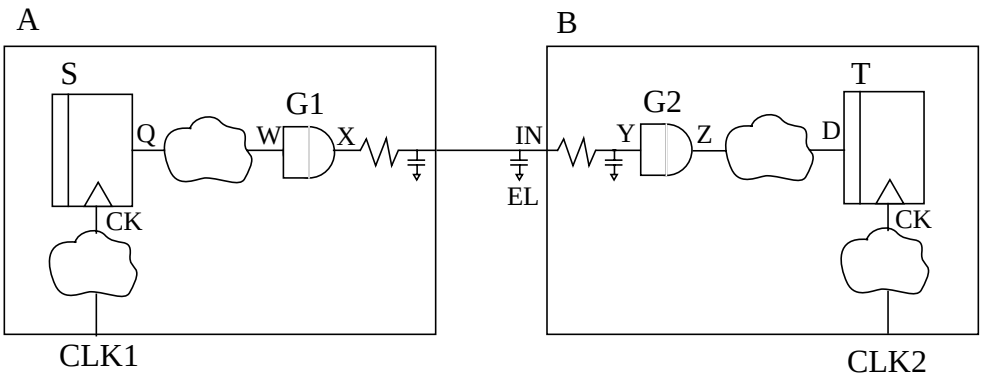
  This is implicitly included in the effective offset of the waveform edge, and should not be included in the *arrival_value* itself.
- The portion of the external insertion delay of the clock network to the source register specified using the **CLOCK_DELAY** construct

  This is implicitly included in the effective offset of the waveform edge, and should not be included in the *arrival_value* itself.
- The CLK->Q delay of the source register
- The delay from the output of the source register up to the input of the driver of the interface net connected to the input pin
- The intrinsic delay of the driver of the interface net.

The delay computed for the partial path starting at the input pin includes

- The load-dependent delay of the driver
- The interconnect delay of the interface net
- The delay from the input of the receiver on the interface net up to the input of the target register
- The setup time (subtracted) or hold time (added) of the target register
- The portion of the insertion delay of the clock network to the target register that is internal to the current GCF cell, specified using the **CLOCK_ARRIVAL** construct (subtracted)
- The portion of the insertion delay of the clock network to the target register that is external to the current GCF cell, specified using the **CLOCK_DELAY** construct (subtracted)

Figure 6 Arrival Time



Example

In Figure 6, the arrival time set on pin IN of block B should include

- A/CLK1 — source clock arrival (implicit in waveform edge)
- A/CLK1 to S/CK — source insertion delay (implicit in waveform edge)
- A/S/CK to A/S/Q — clk->q delay
- A/S/Q to A/G1/W — combinational delay
- A/G1/W to A/G1/X — intrinsic delay

The delay computed for the partial path starting at B/IN includes

- A/G1/W to A/G1/X — load-dependent delay
- A/G1/X to B/G2/Y — interconnect delay
- B/G2/Y to B/T/D — combinational delay
- B/T/D to B/T/CK — setup/hold time
- B/CLK2 to B/T/CK — target insertion delay
- B/CLK2 — target clock arrival

EL is the external load specified on the input pin, and it is included when computing the load-dependent delay and interconnect delay.

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

Example

```
(ENVIRONMENT
  (ARRIVAL (NEGEDGE “100 MHz 50/50”)
    4.0 * 2.0 * A )
)
```

This example specifies the arrival times for the input pin A. Assuming that the time scale is in ns, rise transitions will occur no sooner than 4.0 ns and fall transitions will occur no sooner than 2.0 ns after the falling edge of the reference clock. The latest time at which either rise transitions or fall transitions will occur is unspecified.

Required Time

The **REQUIRED** construct defines ranges of time in which signal transitions must occur at a *port_instance* that includes register data outputs in its transitive fanin. These ranges of time are commonly referred to as required times. Earlier versions of GCF were based on a less commonly used terminology, departure times, which had the same semantics. The **DEPARTURE** keyword is still allowed as a synonymous keyword for the **REQUIRED** construct for backward compatibility.

Required times are usually specified only for primary output and bidirectional ports, but they can also be specified for internal output and bidirectional pins on primitives, and for hierarchical pins on lower level modules or blocks. When specified on internal or hierarchical pins, the required time overrides any propagated required time.

Syntax

required_spec ::= **(** *label? required_keyword*
required_waveform_edge
required_value
*port_instance** **)**

required_keyword ::= **REQUIRED**
||= **DEPARTURE**

required_waveform_edge ::= **(** *waveform_edge_identifier waveform_name* **)**

required_value ::= *target_required_value*

target_required_value ::= *setup_rise_fall hold_rise_fall*
||= (waveform_edge_identifier
setup_value
hold_value) (archaic)

setup_rise_fall ::= *r_rise_fall*

hold_rise_fall ::= *r_rise_fall*

setup_value ::= RNUMBER

hold_value ::= RNUMBER

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

The *required_waveform_edge* specification, which identifies a waveform and an edge of that waveform, is required. The effective offset of the waveform edge implicitly includes any external insertion delay specified for the waveform using the **CLOCK_ARRIVAL** construct, as well as the offset of the edge specified in the waveform definition.

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 required times must have only a single pair of edges.

The required times are target-based, and the *target_required_values* are interpreted as setup and hold constraints.

Specifying a target-based required time is equivalent to adding a register with corresponding setup and hold constraints at the output.

- The hold *required_value* is added to the effective offset of the *required_waveform_edge* in the hold check clock cycle.
- the setup *required_value* is subtracted from the effective offset of the *required_waveform_edge* in the setup check clock cycle.

See “Default Definition” on page 129 for a description of how the setup and hold check clock cycles are normally determined.

All partial paths from the specified *port_instances* to the target registers clocked by a particular waveform edge must be considered in setting the target-based required times related to that waveform edge.

- For the earliest (minimum) required time, the delay of each partial path must be subtracted from the hold time of the target register, and the earliest required 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 earliest required time will usually be a negative number.
- For the latest (maximum) required time, the setup time of the target register must be added to the delay of each partial path, and the latest required time must be set to the largest resulting value.

The *required_value* is a time value and must be specified in the units defined by the *time_scale*. It follows the ordering conventions for *r_rise_fall_min_max* and *r_rise_fall* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

The first *target_required_value* form, *setup_rise_fall hold_rise_fall*, is the preferred form.

The second *target_required_value* form, waveform_edge_identifier setup_value hold_value, is archaic. It is more easily and consistently specified using the first form with asterisks as place-holders.

The required time at an output pin should include

- The delay from the input of the receiver on the interface net up to the input of the target register
- The setup time (subtracted) or hold time (added) of the target register
- The portion of the external insertion delay of the clock network to the target register specified using the **CLOCK_ARRIVAL** construct (subtracted)

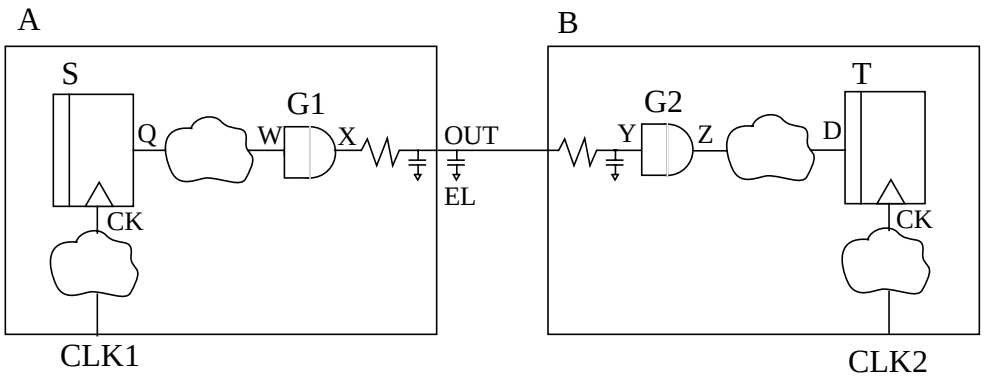
  This is implicitly included in the effective offset of the waveform edge, and should not be included in the *required_value* itself.
- The portion of the external insertion delay of the clock network to the target register specified using the **CLOCK_DELAY** construct (subtracted)

  This is implicitly included in the effective offset of the waveform edge, and should not be included in the *required_value* itself.

The delay computed for the partial path ending at the output pin includes

- The portion of the insertion delay of the clock network to the source register that is internal to the current GCF cell, specified using the **CLOCK_ARRIVAL** construct
- The portion of the insertion delay of the clock network to the source register that is external to the current GCF cell, specified using the **CLOCK_DELAY** construct
- The CLK->Q delay of the source register
- The delay from the output of the source register up to the input of the driver of the interface net connected to the input pin
- The intrinsic delay of the driver of the interface net.
- The load-dependent delay of the driver
- The interconnect delay of the interface net

Figure 7 required Time



Example

In Figure 7, the required time set on pin OUT of block A should include

- B/G2/Y to B/T/D — combinational delay
- B/T/D to B/T/CK — setup/hold time
- B/CLK2 to B/T/CK — target insertion delay (implicit in waveform edge)
- B/CLK2 — target clock arrival (implicit in waveform edge)

The delay computed for the partial path ending at A/OUT includes

- A/CLK1 — source clock arrival
- A/CLK1 to S/CK — source insertion delay
- A/S/CK to A/S/Q — clk->q delay
- A/S/Q to A/G1/W — combinational delay
- A/G1/W to A/G1/X — intrinsic delay
- A/G1/W to A/G1/X — load-dependent delay
- A/G1/X to B/G2/Y — interconnect delay

EL is the external load specified on the output pin, and it is included when computing the load-dependent delay and interconnect delay.

Multiple **REQUIRED** constructs can be defined for the same port. Each **REQUIRED** construct can reference a different *waveform_edge*. The required times associated with a given reference *waveform_edge* are independent of the required 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 **REQUIRED** constructs is cumulative and overriding.

Example

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

This example specifies required 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 that 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 can be specified on both primary interface ports and on internal port instances. If no external delay is specified for a port instances that 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 ::= *r_rise_fall_min_max*
||= (waveform_edge_identifier
r_min_max) (archaic)

The *endpoints_spec* is described in “Path Specifications” on page 110. External delays specified using the **FROM** keyword are to be added to combinational paths that start at the given endpoints, while external delays specified using the **TO** keyword are to be added to combinational paths

that 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 *external_delay_value* is a time value and must be specified in the units defined by the *time_scale*. It follows the ordering conventions for *r_rise_fall_min_max* and *r_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

The *r_rise_fall_min_max* value type is the preferred form for the *external_delay_value*.

The second *external_delay_value* form, waveform_edge_identifier r_min_max, is archaic. It is more easily and consistently specified using the *r_rise_fall_min_max* form with asterisks as place-holders.

The transitions for both forms are with respect to the given endpoints, and the minimum values must be less than or equal to the maximum values for the same transition.

The values specified for external delay should reflect the delay computation on the interface net, which is handled the same as for the **ARRIVAL** and **REQUIRED** constructs.

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

Example

```
(SUBSET TIMING
  (ENVIRONMENT
    (EXTERNAL_DELAY 5.0
      (FROM IN[0] )
    )
    (EXTERNAL_DELAY 3.0 * 2.0 1.0
      (TO OUT[0] )
    )
  )
  (EXCEPTIONS
    (PATH_DELAY 10.0
      ( BETWEEN (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 rise min computed delay of any purely combinational path ending at `OUT[0]`.
- An external combinational delay of 2 ns should be added to the fall min and fall max computed delays of any purely combinational path ending at `OUT[0]`.
- No value is specified for the rise max external delay at `OUT[0]`, so this is not constrained.
- The effective min combinational delay constraint for paths starting at `IN[0]` and ending with a rise transition at `OUT[0]` is 2 ns (the 10 ns **PATH_DELAY** constraint minus external delays of 5.0 and 3.0).
- The effective min combinational delay constraint for paths starting at `IN[0]` and ending with a fall transition at `OUT[0]` is 3 ns (the 10 ns **PATH_DELAY** constraint minus external delays of 5.0 and 2.0).
- The effective max combinational delay constraint for paths starting at `IN[0]` and ending with a fall transition at `OUT[0]` is 4 ns (the 10 ns **PATH_DELAY** constraint minus external delays of 5.0 and 1.0).

Driver Specification

Driver specifications describe information about an external driver that 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 that 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 input slew for a given port always overrides any of the defaults.
- When there are multiple explicit specifications for the same port, the last specification given will be used.
- When there are multiple default specifications, but no explicit specifications for a given port, the last default specified will be used.

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?
*port_instance** **)**

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 that use the convention for *rise_fall_min_max* described in “Value Types” on page 48. They 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.

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.

When a driver cell type is specified on a normal (non-clock) input port, it has three effects on the transition at the inputs of the first stage gates within the current GCF cell:

- The transition is delayed by the load-dependent delay of the driver cell. This does not include the intrinsic delay of the cell.
- The transition is delayed by the interconnect delay, which is computed using the driver cell model in conjunction with the parasitics within the current GCF cell and the external load.
- The effective capacitance of the parasitics is used to determine the slew at the output of the driver. The output slew is then degraded at the loads of the input net to reflect the propagation across the parasitics of the net, including the external load.

The intrinsic delay of the driver cell is defined as the cell delay computed using an output capacitance value of 0 and the input slew(s) specified on the inputs of the driver. The load-dependent delay is the difference between the cell delay computed with the effective capacitance, and the intrinsic delay computed without any load.

For a clock input pin, the driver cell specification is ignored when the nominal values specified by a **CLOCK_DELAY** construct are used instead

of calculating delays. When the nominal **CLOCK_DELAY** values are not used, delays are calculated for clock input pins in the same way as for other pins.

Driver Strength

When the cell type of the external driver is not known, the **DRIVER_STRENGTH** construct can be used instead. Specifying the driver strength is less accurate than specifying the driver cell type, because the effective drive strength for a given cell may vary depending on the load it is driving. When the driver cell type is specified, the effective drive strength can be determined during delay calculation.

Syntax

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

strength_value ::= *rise_fall_min_max*

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

The *strength_value* is a resistance value and must be specified in the units defined by the *res_scale*. It follows the ordering convention for *rise_fall_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

When a driver strength is specified on a normal (non-clock) input pin, it has three effects on the transition at the inputs of the first stage gates within the module:

- The transition is delayed by a “load-dependent delay”, which is modeled as *strength_value* * C_{total}.
- The transition is delayed by the interconnect delay, which is computed using the drive strength in conjunction with the parasitics within the cell and the external load.
- The slew (transition time) at the loads of the input net is also set to *strength_value* * C_{total}. The voltage thresholds for measuring the slew are defined by the **VOLTAGE_THRESHOLD** construct. If no voltage thresholds are specified, the slew represents by default the time required to transition between the 10 and 90 percent points of the power supply voltage.

C_{total} is the sum of the capacitance on the interface net connected to the input pin, including the external load, the internal interconnect capacitance, and the load pin capacitances.

For a clock input pin, the driver strength specification is ignored when the nominal values specified by a **CLOCK_DELAY** construct are used instead of calculating delays. When the nominal **CLOCK_DELAY** values are not used, delays are calculated for clock input pins in the same way as for other pins.

Input Slew

When the cell type of the external driver is not known, the **INPUT_SLEW** construct can be used instead. In general, the **INPUT_SLEW** construct should only be used for primary inputs on a chip. For chip-level inputs, the **DRIVER_CELL** and **DRIVER_STRENGTH** constructs would be less accurate than **INPUT_SLEW**, because the on-chip delay modeling would not properly account for the effects of the board-level interconnect, package pins, etc.

For inputs on modules within a chip, the **DRIVER_CELL** construct is more accurate than **INPUT_SLEW**, because it takes into account the interaction between the driver and the interconnect.

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*. It follows the convention for *rise_fall_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

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.

When an input slew specified on a normal (non-clock) input pin, it has two effects on the transition at the inputs of the first stage gates within the module:

- The transition is delayed by the interconnect delay, which is computed using an artificial driver model (with zero delay and a fixed output slew equal to the specified input slew) in conjunction with the parasitics within the cell and the external load.
- The specified input slew is degraded at the loads of the input net to reflect the propagation across the parasitics of the net, including the external load.

There is no modeling of the load-dependent delay of the driver.

For a clock input pin, the input slew specification is ignored when the nominal values specified by a **CLOCK_DELAY** construct are used instead of calculating delays. When the nominal **CLOCK_DELAY** values are not used, delays are calculated for clock input pins in the same way as for other pins.

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** *slew_value* *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 *slew_value* is a time value and must be specified in the units defined by the *time_scale*. It follows the convention for *rise_fall_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

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.

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 required 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.

Archaic timing exception constructs are described starting on page 158.

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*
||= *slew_limit_spec*
||= max_transition_time_spec (archaic)
||= *extension*

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_mode_spec*
||= *clock_delay_spec*
||= *clock_uncertainty_spec*
||= *meta_data_1*

Path Specifications

Many of the timing exceptions require path specifications. This section describes ways of specifying paths that are common to several types of timing exceptions.

Level 0 THRU Specifications

The Level 0 *thru_spec* construct constrains all paths that pass through a single *port_instance*, including those that start or end at the *port_instance*.

Syntax

thru_spec ::= **(THRU** *port_instance* **)**

When specified on a flip flop data output, the *thru_spec* construct affects paths from the flip flop clock input through the output, and paths through asynchronous preset and clear inputs through the output).

When specified on a flip flop data input, the *thru_spec* construct affects paths ending at the data input.

When specified on a flip flop clock input, the *thru_spec* construct affects paths from the clock input. It does not affect paths ending at the flip flop data input.

When specified on a latch data output, the *thru_spec* construct affects paths from the latch enable input through the output, paths through the latch data input through the output, and paths through asynchronous preset and clear inputs through the output). It does not affect borrowing for paths ending at the latch data input.

When specified on a latch data input, the *thru_spec* construct affects paths ending at the data input and paths through the data input through the output.

When specified on a latch enable input, the *thru_spec* construct affects paths from the enable input. It does not affect paths ending at the latch data input or paths through the data input through the data output.

Example

```
(THRU ff1.Q)
```

This example constrains all paths that go through the Q output of the flip flop ff1. This includes paths from the clock input through the output and paths from asynchronous preset or clear inputs through the output.

Level 0 *port_instance* Specifications

When a *port_instance* is specified by itself (which is only used in the **DISABLE** construct), the semantics are different than if the same *port_instance* was specified within a *thru_spec*, *from_spec*, or *to_spec*.

Starting with GCF 1.4, when a *port_instance* is specified by itself within a **DISABLE** construct, it disables

- slews that would otherwise propagate through the *port_instance* during delay calculation
- constants that would otherwise propagate through the *port_instance* during delay calculation
- timing checks on paths that pass through the *port_instance* during timing analysis.

When a *port_instance* is specified within a *thru_spec*, *from_spec*, or *to_spec* **DISABLE**, only the timing checks are disabled. In earlier versions of GCF, this was also true for the case where a *port_instance* was specified by itself within a **DISABLE** construct.

When a *port_instance* on a combinational gate is specified, it affects all paths through the *port_instance*.

When a flip flop data output is specified, it affects paths from the related flip flop clock input through the output, as well as paths through asynchronous preset and clear inputs through the output.

When a flip flop data input is specified, it does not affect paths ending at the data input.

When a flip flop clock input is specified, it affects all paths to, from, or through the flip flop.

When a latch data output is specified, it affects paths from the related latch enable input through the output, paths through the latch data input through the output, and paths through asynchronous preset and clear inputs through the output. It also affects borrowing for paths ending at the latch data input.

When a latch data input is specified, it disables borrowing between the paths ending at the data input and paths starting at the latch enable, but it does not otherwise affect those paths.

When a latch enable input is specified, it affects all paths to, from, or through the latch.

Level 0 *cell_instance* Specifications

When a *cell_instance* is specified by itself (which is only used in the **DISABLE** construct), the semantics are different than if the same *cell_instance* was specified within a *from_spec*, *to_spec*, or *disable_instance_spec*.

Starting with GCF 1.4, when a *cell_instance* is specified by itself within a **DISABLE** construct, it disables

- slews that would otherwise propagate through the affected pins of the *cell_instance* during delay calculation, and
- constants that would otherwise propagate through the affected pins of the *cell_instance* during delay calculation
- timing checks on paths that pass through the *cell_instance* during timing analysis.

When a *cell_instance* is specified within a *thru_spec*, *from_spec*, or *to_spec* **DISABLE**, only the timing checks are disabled. In earlier versions of GCF, this was also true for the case where a *cell_instance* was specified by itself within a **DISABLE** construct.

When a *cell_instance* that is a combinational gate is specified, it affects all paths through the *cell_instance*.

When a flip flop is specified, it disables all paths to, from, or through the flip flop.

When a latch is specified, it affects paths from the related latch enable input through the output, paths through the latch data input through the output, and paths through asynchronous preset and clear inputs through the output. It also affects borrowing for paths ending at the latch data input.

Level 0 Arc Specifications

The Level 0 **ARC** construct constrains all paths that pass through a pair of *port_instances*, including paths that 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.

When the starting point of the arc is an output on a flip flop or latch, the same paths as a *thru_spec* construct with the same *port_instance* are considered, and only those paths that also pass through the ending point of the arc are specified.

When the ending point of the arc is an input on a flip flop or latch, the same paths as a *thru_spec* construct with the same *port_instance* are considered, and only those paths that also pass through the starting point of the arc are specified.

Starting with GCF 1.4, when an *arc_spec* is specified within a **DISABLE** construct, it disables

- slews that would otherwise propagate through the arc during delay calculation, and
- constants that would otherwise propagate through the arc during delay calculation
- timing checks on paths that pass through the arc during timing analysis.

Syntax

arc_spec ::= **(ARC** *port_instance port_instance* **)**

Example

```
(ARC or1.a or1.z)
```

This example constrains all paths that go through the A input and the Z output of *or1*.

Example

```
(ARC ff1.clk ff1.q)
```

This example constrains all paths that start at the clock input and go through the *q* output of *ff1*. If there is an inverting output, *qn*, paths through it are not affected.

Level 0 Endpoint Specifications

The Level 0 *endpoints_spec* construct specifies paths in terms of their endpoints.

Syntax

```
endpoints_spec ::= from_spec
               ||= to_spec
               ||= ( BETWEEN? 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
               ||= typed_waveform_name_list
               ||= typed_port_expr
               ||= typed_pin_expr
               ||= typed_instance_expr
```

If only **FROM** items are specified, they refer to a set of starting points for paths, and all paths that start at any of those points and end at either register data inputs or primary output/bidirectional ports are constrained.

If only **TO** items are specified, they refer to a set of ending points for paths, and all paths that end at any of those points and start at either register clock inputs or primary input/bidirectional ports are constrained.

If both **FROM** and **TO** items are specified using the **BETWEEN** form, they refer to a set of starting points and endpoint points for paths, and all paths between any of those starting points and any of those ending points are constrained. The **BETWEEN** keyword is optional for backward compatibility with earlier versions of GCF, but it can be included for clarity.

Some of the exception constructs allow multiple *endpoint_specs*. In this case, if both **FROM** and **TO** items are specified in different *endpoint_specs* (without using the **BETWEEN** form), the effect is to constrain all paths from the starting points, as well as (separately) all paths to the ending points. This will generally constrain more paths than specifying the same **FROM** and **TO** items within a **BETWEEN** construct.

Disable And Multi-Cycle Endpoint Specifications

See “Disabling Paths Between Endpoints” on page 125, and “Multi-Cycle Paths Between Endpoints” on page 131 for details on how the *endpoints_spec* construct is used in these cases.

FROM items used in an *endpoints_spec* for a Level 0 **DISABLE** or **MULTI_CYCLE** construct must be waveform names, primary input or

bidirectional ports, registers, register clock inputs, or register data outputs. A *port_instance* on an intermediate level of hierarchy may also be specified as a **FROM** item, when the internal net is driven directly by a register.

Combinational *port_instances* or *cell_instances* are not allowed as **FROM** items in GCF 1.4.

When waveform names, registers, or *port_instances* on intermediate levels of hierarchy are specified as **FROM** items, they implicitly refer to a set of register clock inputs, and/or a set of primary input or bidirectional ports, which are the actual starting points for the paths described by the *endpoints_spec*.

- For a primary input or bidirectional port, the port itself is the the starting point.
- For a register clock input, the input itself is the starting point.
- For a waveform name, the starting points are the register clock inputs in the transitive fanout of each clock pin to which the waveform is assigned, as well as any primary input or bidirectional ports that have an arrival time referenced to that waveform.
- For a register name, the starting points are the clock inputs on that register.
- For a *port_instance* on an intermediate level of hierarchy, the starting point is the clock input on the register that drives the internal net.

When a *typed_port_expr*, *typed_pin_expr*, or *typed_instance_expr* is used as a **FROM** item, the expression is expanded to refer to the set of ports, pins, or instances that both match the expression and would also be legal **FROM** items for the constraint.

TO items used in an *endpoints_spec* for a Level 0 **DISABLE** or **MULTI_CYCLE** construct must be waveform names, primary output or bidirectional ports, registers, register data inputs, or register clock inputs. A *port_instance* on an intermediate level of hierarchy may also be specified as a **TO** item, when the internal net directly drives a register.

Combinational *port_instances* or *cell_instances* are not allowed as **TO** items in GCF 1.4.

When waveform names, registers, or *port_instances* on intermediate levels of hierarchy are specified as **TO** items, they implicitly refer to a set of register data inputs, and/or a set of primary output or bidirectional ports,

which are the actual ending points for the paths described by the *endpoints_spec*.

- For a primary output or bidirectional port, the port itself is the the ending point.
- For a register data input, the input itself is the ending point.
- For a waveform name, the ending points are the register data inputs on registers whose clock input is in the transitive fanout of each clock pin to which the waveform is assigned, as well as any primary output or bidirectional ports that have a required time referenced to that waveform.
- For a register name, the ending points are the data inputs on that register.
- For a *port_instance* on an intermediate level of hierarchy, the ending point is the clock input on the register that is driven by the internal net.

When a *typed_port_expr*, *typed_pin_expr*, or *typed_instance_expr* is used as a **TO** item, the expression is expanded to refer to the set of ports, pins, or instances that both match the expression and would also be legal **TO** items for the constraint.

Path Delay Endpoint Specifications

See “Combinational Delays” on page 135 for details on how the *endpoints_spec* construct is used with the **PATH_DELAY** construct.

FROM items used in an *endpoints_spec* for a Level 0 **PATH_DELAY** construct can be any of the types that are allowed for the **FALSE** and **MULTI_CYCLE** constructs, with the same rules for implicitly determining the actual starting points for the constrained paths. In addition, **FROM** items can be internal port instances on combinational logic (input, output, or bidirectional) or output port instances on registers.

TO items used in an *endpoints_spec* for a Level 0 **PATH_DELAY** construct can be any of the types that are allowed for the **FALSE** and **MULTI_CYCLE** constructs, with the same rules for implicitly determining the actual ending points for the constrained paths. In addition, **TO** items can be internal port instances on combinational logic (input, output, or bidirectional).

Level 0 From, To, Thru Specification

The Level 0 *from_to_thru_spec* construct constrains paths that start at the **FROM** endpoints (if given), pass through the **THRU_ALL** points (if given), and end at the **TO** endpoints (if given). The affected transitions through the constrained paths can be specified using edges at each point.

Syntax

from_to_thru_spec ::= **(PATHS** *from_to_thru_item+* **)**

from_to_thru_item ::= *from_opt_edge_spec*
||= *to_opt_edge_spec*
||= *thru_all_items_spec*

from_opt_edge_spec ::= *from_spec*
||= **(FROM** *from_item_edge+* **)**

to_opt_edge_spec ::= *to_spec*
||= **(TO** *to_item_edge+* **)**

from_item_edge ::= **(** *edge_identifier from_to_item+* **)**

to_item_edge ::= **(** *edge_identifier from_to_item+* **)**

thru_all_items_spec ::= **(THRU_ALL**
thru_any_item_spec+ **)**

thru_any_item_spec ::= *thru_item*
||= **(THRU_ANY** *thru_item+* **)**

thru_item ::= *port_instance*
||= *net*
||= *typed_port_expr*
||= *typed_pin_expr*
||= *typed_net_expr*
||= *port_instance_edge*

port_instance_edge ::= **(** *edge_identifier port_instance* **)**

The *from_to_thru_spec* must include at most one of each type of *from_to_thru_item* (one set of **FROM** endpoints, one set of **THRU_ALL** points, and/or one set of **TO** endpoints).

In the *from_opt_edge_spec* and the *to_opt_edge_spec*, the **FROM** and **TO** endpoints follow the same conventions as in the Level 0 *endpoints_spec*. See “Level 0 Endpoint Specifications” on page 114 for details on the types of endpoints that are allowed and the paths that are constrained.

When an *edge_identifier* is specified for the *from_opt_edge_spec*, only that transition through each of the constrained paths is affected.

When an *edge_identifier* is specified for the *to_opt_edge_spec*, only that transition at the endpoint of each constrained path is affected.

When a *typed_port_expr*, *typed_pin_expr*, or *typed_net_expr* is used as a *thru_item*, the expression is expanded in place to refer to the set of ports, pins, or nets that match the expression. Therefore, the expression affects paths that go **THRU_ANY** one of the ports, pins, or nets, not paths that go **THRU_ALL** of the ports, pins, or nets.

When a *port_instance_edge* is specified for a *thru_item*, only the transitions through each of the constrained paths that result in the specified edge at that *port_instance* are affected.

If the *thru_all_items_spec* is given, each constrained path must go through at least one of the *thru_items* listed in each of the *thru_any_item_spec*s, in the order in which the *thru_any_items_specs* are listed. The *thru_items* in the *thru_any_item_specs* do not have to be contiguous in the paths.

Example

```
(PATHS
  (FROM ff1.clk)
  (THRU_ALL
    ff1.q
    and1.a
  )
  (TO ff2.d)
)
```

This example constrains all paths that start at either *ff1.clk*, go through *ff1.q* followed by *and1.a*, then end at *ff2.d*. All transitions through the constrained paths are affected.

Example

```
(PATHS
  (THRU_ALL
    and1.a
  )
)
```

This example constrains all transitions through all paths that go through *and1.a*.

Example

```
(PATHS
  (THRU_ALL
    and2
    (posedge and3.a)
    (negedge and4.a)
  )
)
```

This example constrains all paths that go through the cell instance *and2*, followed by *and3.a*, then *and4.a*. Only the transitions through the constrained paths that result in a rising edge at *and3.a* and a falling edge at *and4.a* are affected.

Example

```
(PATHS
  (FROM (negedge in1 in2))
  (THRU_ALL
    and1.a
    (posedge and2.a)
    (THRU_ANY and3.a (negedge and4.a))
  )
  (TO (posedge ff1.d ff2.d))
)
```

This example constrains all paths that start at either *in1* or *in2*, go through *and1.a* followed by *and2.a*, then go through either *and3.a* or *and4.a*, then end at either *ff1.d* or *ff2.d*.

Only certain transitions through the constrained paths are affected:

- a falling transition at *in1* or *in2* that results in a
- rising transition at *and2.a* that results in a
- falling transition at *and4.a* (or either transition at *and3.a*) that results in a
- rising transition at *ff1.d* or *ff2.d*.

Precedence Rules for Exceptions

In general, **DISABLE**, **MULTI_CYCLE**, and **PATH_DELAY** exceptions should be viewed as modifying the default analysis based on either the early (minimum) delay or the late (maximum) delay of a particular transition propagating through a particular path in the circuit.

The following precedence rules are used in the order given when several exceptions affect the analysis for the same type of delay (early or late) for the same transition propagating through the same path in the circuit:

- **DISABLE** has the highest precedence
- **PATH_DELAY** has higher precedence than **MULTI_CYCLE**
- A **PATH_DELAY** (**MULTI_CYCLE**) construct that includes a *port_instance* as a *from_item* has higher precedence than a **PATH_DELAY** (**MULTI_CYCLE**) construct that does not.
- A **PATH_DELAY** (**MULTI_CYCLE**) construct that includes a *port_instance* as a *to_item* has higher precedence than a **PATH_DELAY** (**MULTI_CYCLE**) construct that does not.
- A **PATH_DELAY** (**MULTI_CYCLE**) construct that includes a *thru_all* specification has higher precedence than a **PATH_DELAY** (**MULTI_CYCLE**) construct that does not.
- A **PATH_DELAY** (**MULTI_CYCLE**) construct that includes a *waveform_name* as a *from_item* has higher precedence than a **PATH_DELAY** (**MULTI_CYCLE**) construct that does not.
- A **PATH_DELAY** (**MULTI_CYCLE**) construct that includes a *waveform_name* as a *to_item* has higher precedence than a **PATH_DELAY** (**MULTI_CYCLE**) construct that does not.
- The **PATH_DELAY** or **MULTI_CYCLE** construct that specifies the tightest constraint on the delay of the transition through the path is used.

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 early (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 early timing checks.
- The late (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 late timing checks.

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

Slew Propagation and Disables

In GCF 1.4 and above, some types of disables affect slew propagation. Normally during delay calculation and timing analysis, the slew at a pin is computed from the slew propagated through the timing arcs that end at the pin, and the slew at the pin is then propagated through the timing arcs that start at the pin. When a pin is specified in the *port_instance* form or an output pin is implied by the *cell_instance* form of *disable_item_spec_0*, slews will not be propagated through the timing arcs that start at the pin.

When the *arc_spec* form of *disable_item_spec_0* is used, slews will not be propagated through any timing arcs between the endpoints of the *arc_spec*.

Similarly, when preset and clear arcs on registers are disabled using *preset_clear_spec*, the slews at the preset or clear input pin will not be propagated through the timing arcs that start at the pin.

Other types of disables, such as the *reentrant_paths_spec* and the *disable_from_to_thru_spec*, do not affect slew propagation during delay calculation for SDF generation. The effect of these other types of disables on slew propagation during the delay calculation used for timing analysis is unspecified in GCF 1.4; some tools may disable slew propagation in these cases, while other tools may not. The behavior is expected to be explicitly specified in a subsequent version of GCF.

Constant Propagation an Disables

In GCF 1.4 and above, some types of disables affect constant propagation. Normally, constant values specified using the **CONSTANT** construct are propagated through the circuit, causing additional pins to have a constant value. When a pin is specified in the *port_instance* form or an output pin is implied by the *cell_instance* form of *disable_item_spec_0*, constant values will not be propagated through the pin.

When the *arc_spec* form of *disable_item_spec_0* is used, constants will not be propagated through any timing arcs between the endpoints of the *arc_spec*.

Similarly, when preset and clear arcs on registers are disabled using *preset_clear_spec*, constants will not be propagated through the timing arcs that start at the pin.

Other types of disables do not affect constant propagation.

Level 0 Disables

In Level 0, the paths can be identified by a single port instance, a cell instance, the path endpoints, or by a set of from, to, and through items.

Syntax

```
disable_spec_0 ::= disable_item_spec_0
               ||= disable_endpoints_spec_0
               ||= disable_from_to_thru_spec_0
```

The simplest form of the **DISABLE** construct, *disable_item_spec_0*, disables all timing checks associated with a set of paths, and in some cases, slew propagation and constant propagation along those paths.

Syntax

```
disable_item_spec_0 ::= ( label? DISABLE disable_item_0+ )
      disable_item_0 ::= port_instance
                     ||= cell_instance
                     ||= typed_port_expr
                     ||= typed_pin_expr
                     ||= typed_instance_expr
                     ||= arc_spec
                     ||= preset_clear_spec
                     ||= reentrant_paths_spec
  preset_clear_spec ::= ( PRESET_CLEAR_ARCS true_false )
reentrant_paths_spec ::= ( REENTRANT_PATHS true_false )
        true_false ::= TRUE
                   ||= FALSE
```

Disabling Port Instances, Cell Instances, and Arcs

All timing checks associated with the constrained paths are disabled. For details on the paths constrained by the *port_instance*, *cell_instance*, and *arc_spec* forms, refer to “Path Specifications” on page 110.

Note that the Level 0 *cell_instance* form is not the same as the Level 1 *disable_instance_spec*, which disables all paths associated with the instance (from, to, or through). The *cell_instance* form only disables paths through the outputs.

Example

```
(DISABLE ff1.Q)
```

This example disables all paths that go through the Q output of the flip flop ff1. This includes paths from the clock input through the output and paths from asynchronous preset or clear inputs through the output.

Example

```
(DISABLE ff1)
```

This example constrains all paths that go through the data outputs of the flip flop ff1.

Example

```
(DISABLE (ARC or1.A or1.Z))
```

This example disables all paths that go through the A input and the Z output of or1.

Disabling Paths Through Asynchronous Preset and Clear Arcs

If the *preset_clear_spec* construct is used, it specifies whether delay arcs starting at preset or clear inputs on registers in the current GCF cell should be disabled. Disabling these arcs suppresses paths that go through preset or clear inputs, but does not suppress timing checks associated with those inputs.

When applied to a non-leaf GCF cell, the *preset_clear_spec* setting applies to all registers within that cell and all of its descendents, unless overridden by a *preset_clear_spec* for one of the descendents.

The default is that the delay arcs starting at preset and clear inputs are disabled.

Example

```
(DISABLE (PRESET_CLEAR_ARCS FALSE))
```

This example enables paths that go through asynchronous preset and clear inputs.

Disabling Reentrant Bidirectional Paths

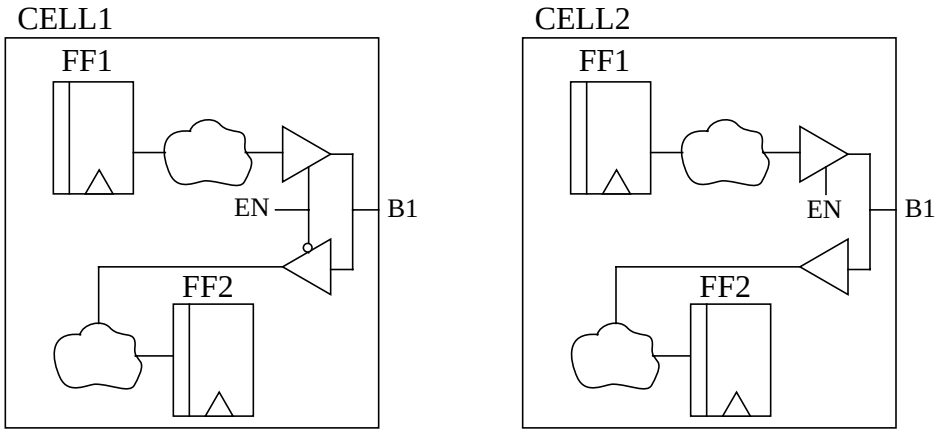
If the *reentrant_paths_spec* construct is given, it specifies whether reentrant bidirectional paths should be disabled within the current GCF cell. By default, these reentrant paths are disabled.

There are two types of reentrant bidirectional paths:

- Paths through nets connected to primary bidirectional ports on the current GCF cell, as shown in Figure 8.

- Paths through primitives that have a bidirectional pin, where the timing arcs to and from the bidirectional pin form a reentrant connection, as shown in Figure 9.

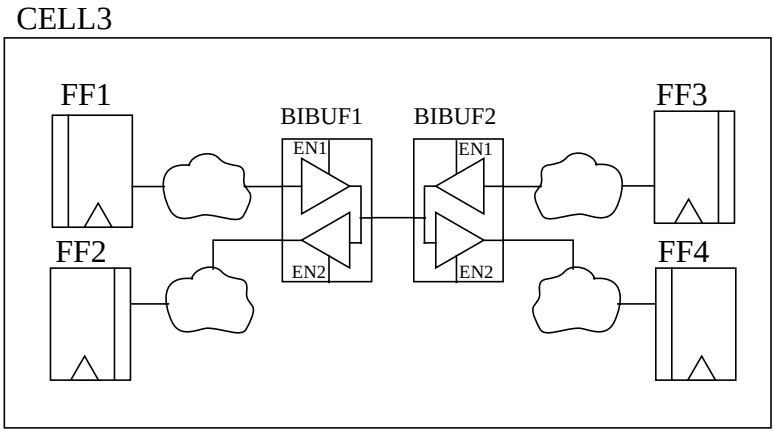
Figure 8 Reentrant Paths for Primary Bidirectional Ports



In Figure 8, B1 is a primary bidirectional port for both CELL1 and CELL2. Normally, for a bidirectional port like B1, there will be paths from source registers within the cell to target registers outside the cell, as well as paths from source registers outside the cell to target registers inside the cell. The *reentrant_paths_spec* disable affects reentrant paths, where both the source and target registers are within the cell.

In both CELL1 and CELL2, the paths from FF1 to FF2 are reentrant. Whether it makes sense to analyze the timing of these paths depends on the design style. In CELL1, using complementary enables on the tri-state buffers ensures that the paths from FF1 to FF2 can never be activated. For this design style, reentrant paths should be disabled. In CELL2, the paths from FF1 to FF2 can be activated, so reentrant paths should not be disabled.

Figure 9 Reentrant Paths Through a Bidirectional Primitive



In Figure 9, whether analyzing the reentrant path from FF1 to FF2 through the bidirectional primitive BIBUF1 makes sense depends on the design style. If complementary enables are connected to EN1 and EN2 on BIBUF1, then the reentrant path will never be active. The reentrant path from FF3 to FF4 through BIBUF2 is similar. The paths from FF1 to FF4, and from FF3 to FF2 are not reentrant, and are not affected by the *reentrant_paths_spec*.

Example

```
(DISABLE (REENTRANT_PATHS FALSE))
```

This example enables reentrant paths.

Disabling Paths Between Endpoints

The *disable_endpoints_spec_0* construct disables selected timing checks on a set of paths that are identified by their from, to, or both from and to endpoints. See “Level 0 Endpoint Specifications” on page 114 for details on the types of endpoints that are allowed and the paths that are affected.

Syntax

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

disable_option ::= *timing_check*
||= *edge_identifier*

If the **HOLD** or **SETUP** keyword is specified as a *disable_option*, only the early (minimum) or late (maximum) timing checks will be disabled; otherwise, both the early and late timing checks will be disabled.

If an *edge_identifier* (for example, **POSEDGE** or **NEGEDGE)** is specified as a *disable_option*, only the timing checks on the rising or falling data

transitions at the path target will be disabled. Otherwise, both the rising and falling timing checks will be disabled.

Example

```
(DISABLE
  ( BETWEEN (FROM ff1.clk) (TO ff2.d ff3.d) )
)
```

The disable specification in this example affects all paths between *ff1* and either *ff2* or *ff3*.

Example

```
(DISABLE
  (FROM ff1.clk) setup posedge
)
```

This example disables setup checks for the rising edge at the target for all paths starting at *ff1.clk*.

Disabling Paths With From, To, and Thru

The *disable_from_to_thru_spec_0* construct disables all paths that start at the **FROM** endpoints (if given), pass through the **THRU_ALL** points (if given), and end at the **TO** endpoints (if given). See “Level 0 From, To, Thru Specification” on page 117 for details on the types of items that can be specified for *from_to_thru_spec* and the paths that are affected.

Syntax

disable_from_to_thru_spec_0 ::= **(** *label?* **DISABLE**
*from_to_thru_spec+ disable_option** **)**

If the **HOLD** or **SETUP** keyword is specified as a *disable_option*, only the early (minimum) or late (maximum) timing checks will be disabled. Otherwise, both the late and early timing checks will be disabled.

If an *edge_identifier* (for example, **POSEDGE** or **NEGEDGE)** is specified as a *disable_option*, only the timing checks on the rising or falling data transitions at the path target will be disabled. Otherwise, both the rising and falling timing checks will be disabled.

The *edge_identifier* as a *disable_option* applies in general to all of the endpoints implied by the *from_to_thru_spec*. The general *edge_identifier* has a lower precedence than an *edge_identifier* specified explicitly for a particular *to_item* within the *from_to_thru_spec*.

Example

```
(DISABLE
  (PATHS
    (FROM ff1.clk)
    (THRU_ALL
      (THRU_ANY and1.a and2.a)
      (THRU_ANY and3.a and.a)
      mux.a
    )
    (TO ff2.d)
  )
)
```

This example disables all paths that start at *ff1.clk*, go through either *and1.a* or *and2.a*, followed by *and3.a* or *and4.a*, followed by *mux.a*, then end at *ff2.d*. The intermediate thru points do not have to be contiguous in the path.

Example

```
(DISABLE
  (PATHS
    (FROM (posedge in1))
    (THRU_ALL
      (THRU_ANY and1.a and2.a)
      and3.z
    )
    (TO ff2)
  )
)
```

This example disables the rising transition propagating through all paths that start at *in1*, go through either *and1.a* or *and2.a*, followed by *and3.z*, then end at *ff2*. The intermediate thru points do not have to be contiguous in the path.

Example

```
(DISABLE
  (PATHS
    (FROM (posedge ff1.clk)
    (THRU_ALL
      and1.a
    )
    (TO ff2.d)
  )
  hold
  negedge
)
```

This example disables early timing checks for all paths that start at *ff1.clk* and go through *and1.a*, for the transitions through each path that start with a rising transition on the clock at *ff1.clk* and end with a falling data transition at *ff2.d*.

Level 1 Disables

In Level 1, there are several additional ways in which to specify paths that are to be disabled.

Syntax

```
disable_spec_1 ::= disable_cell_spec_1
               ||= disable_edges_spec_1        (archaic)
```

Disabling Cell Instances and Cell Types

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.

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 untyped_cell_instance+ )
disable_master_spec ::= ( MASTER cell_id )
```

Example

```
(DISABLE (INSTANCE vco))
```

This example disables all paths associated with the *vco* instance.

Example

```
(DISABLE (MASTER (CELLTYPE DUMMY))
```

This example disables all paths associated with all occurrences of the *DUMMY* cell.

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 that 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, multi-cycle paths can be identified by the path endpoints, or by a set of from, to, and through items.

Syntax

```
multi_cycle_spec_0 ::= multi_cycle_endpoints_spec_ 0
                   ||= multi_cycle_from_to_thru_spec_0
```

Multi-Cycle Paths Between Endpoints

See “Level 0 Endpoint Specifications” on page 114 for a description of the types of endpoints that are allowed and the paths that are affected.

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_endpoints_spec_0 ::= ( label? MULTI_CYCLE
                                   multi_cycle_endpoints_param_list )
multi_cycle_endpoints_param_list ::= multi_cycle_option+ endpoints_spec+
                                 ||= endpoints_spec+ multi_cycle_option+
multi_cycle_option ::= timing_check_offset
                   ||= edge_identifier
timing_check_offset ::= ( timing_check num_cycles
                          reference_clock? )
reference_clock ::= SOURCE
                ||= TARGET
num_cycles ::= INUMBER
```

At least one *timing_check_offset* and at least one *endpoints_spec* must be specified in the *multi_cycle_endpoints_param_list*.

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 that 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 that 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 that 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
      ( BETWEEN
        (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 between registers whose source clock originates at *clk1* and registers whose target clock originates at *divider.clkout*.

- 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)
  ( BETWEEN (FROM ff1.clk) (TO ff2.d ff3.d) )
)
```

The multi-cycle path specification in this example has the following effects on all paths that start at *ff1* and end at either *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.

Multi-Cycle Paths With From, To, and Thru

The *multi_cycle_from_to_thru_spec_0* construct constrains all paths that start at the **FROM** endpoints (if given), pass through the **THRU_ALL** points (if given), and end at the **TO** endpoints (if given). See “Level 0 From, To, Thru Specification” on page 117 for details on the types of items that can be specified for *from_to_thru_spec* and the paths that are affected.

Syntax

multi_cycle_from_to_thru_spec_0 ::= **(** *label?* **MULTI_CYCLE** *multi_cycle_from_to_thru_param_list* **)**

multi_cycle_from_to_thru_param_list ::= *from_to_thru_spec+ multi_cycle_option+*
||= *multi_cycle_option from_to_thru_spec+*

At least one *timing_check_offset* and at least one *from_to_thru_spec* must be specified in the *multi_cycle_from_to_thru_param_list*.

Example

```
(MULTI_CYCLE
  (SETUP 2)
  (PATHS
    (FROM ff1)
    (THRU_ALL
      (posedge and1.a)
    )
    (TO ff2)
  )
)
```

The multi-cycle path specification in this example has the following effects on all paths that start at *ff1*, go through *and1.a*, and end at *ff2* or *ff3*:

- For the setup check on the data edge(s) at the target that result from a rising transition at *and1.a*, 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 the data edge(s) at the target that result from a rising transition at *and1.a* 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

```
(MULTI_CYCLE
  (SETUP 2 SOURCE)
  (PATHS
    (FROM (posedge in1))
    (THRU_ALL
      (THRU_ANY and1.a and2.a)
      and3.z
    )
    (TO ff2)
  )
)
```

The multi-cycle path specification in this example has the following effects on all paths that start at *in1*, go through either *and1.a* or *and2.a*, followed by *and3.z*, then end at *ff2*:

- For the setup check on the data edge(s) at the target that result from a rising transition at *in1*, the active edge at the source is one source clock cycle earlier than the default. The default active edge at the target is unchanged.
- The hold check on the data edge(s) at the target that result from a rising transition at *and1.a* 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.

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

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 that 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.

Some forms of the **PATH_DELAY** construct allow the starting points for the constrained paths to be unspecified. When the starting points are unspecified, all paths that start at a register clock input or a primary input/bidirectional port instance and go through the other logic specified in the **PATH_DELAY** construct are constrained.

Some forms of the **PATH_DELAY** construct allow the ending points for the constrained paths to be unspecified. When the ending points are unspecified, all paths that go through the other logic specified in the **PATH_DELAY** construct and end at a register data input or a primary output/bidirectional port instance are constrained.

The paths through combinational logic constrained by a **PATH_DELAY** constructs cannot go through a data input on a latch, through a transparent arc, then through a data output. However, the constrained paths can start at a clock input on a flip flop or an enable input on a latch, then go through a data output. When **PRESET_CLEAR_ARCS** are enabled (see page 123), the constrained paths can also go through an asynchronous preset or clear input, then through a data output.

The *path_delay_value* is a time value and must be specified in the units defined by the *time_scale*. It follows the ordering conventions for *rise_fall_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

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

- For paths starting at clock inputs or data outputs of a sequential element, the clock insertion delay to the sequential element will be added to the combinational delay of the path before comparing against the constraint.

  For early (minimum) delay constraints, the minimum clock insertion delay will be added.

For late (maximum) delay constraints, the maximum clock insertion delay will be added.

- For paths ending at data inputs of a sequential element, the clock insertion delay to the sequential element will be added to the constraint value before comparing against the constraint.

  For early (minimum) delay constraints, the maximum clock insertion delay will be added.

  For late (maximum) delay constraints, the minimum clock insertion delay will be added.

- For paths ending at data inputs of a sequential element, the setup and hold times will be used to adjust the constraint value before comparing against the constraint.

  For early (minimum) delay constraints, the hold time will be added.

  For late (maximum) delay constraints, the setup time will be subtracted.

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 142).

Level 0 Combinational Path Delays

In Level 0, combinational path delays can be identified by the path endpoints, or by a set of from, to, and through items.

Syntax

```
path_delay_spec_0  ::= path_delay_endpoints_spec_ 0
                   ||= path_delay_from_to_thru_spec_0
```

Path Delays Between Endpoints

See “Level 0 Endpoint Specifications” on page 114 for a description of the types of endpoints that are allowed and the paths that are affected.

Syntax

```
path_delay_endpoints_spec_0  ::= ( label? PATH_DELAY
                                     path_delay_value
                                     endpoints_spec+ )
          path_delay_value   ::= rise_fall_min_max
                             ||= path_delay_single_value(archaic)
path_delay_single_value      ::= ( timing_check
                                     waveform_edge_identifier
                                     NUMBER )(archaic)
```

The *rise_fall_min_max* value type is the preferred form for the *path_delay_value*.

The second form, path_delay_single_value, is archaic. It is more easily and consistently specified using the *rise_fall_min_max* form with asterisks as place-holders.

Example

```
(PATH_DELAY 4.0 5.5 * *
  (BETWEEN
    (FROM scan_di)
    (TO ff1.d)
  )
)
```

This example specifies the combinational delay for certain transitions through all paths between *scan_di* and *ff1.d*. Because *ff1.d* is an input on a sequential element, the effects of clock skew and timing checks will be considered in the constraint.

Suppose that ideal clock insertion delays have been defined such that the minimum clock insertion delay to *ff1* is 1.0 ns, the maximum clock insertion delay is 1.3 ns, the hold time for the rising transition at *ff1.d* is 0.4 ns, and the setup time for the rising transition at *ff1*.d is 0.6 ns.

In that case,

- the effective early (minimum) delay constraint is 4.0 + 1.0 + 0.4 = 5.4 ns
- the effective late (maximum) delay constraint is 5.5 + 1.3 - 0.6 = 6.2 ns.
- These constraints affect only the transitions through the constrained paths that result in a rising transition at *ff1.d*. The delay for other transitions is unconstrained.

Path Delays With From, To, and Thru

The *path_delay_from_to_thru_spec_0* construct constrains all paths that start at the **FROM** endpoints, pass through the **THRU** points, and end at the **TO** endpoints. See “Level 0 From, To, Thru Specification” on page 117 for details on the types of items that can be specified for *from_to_thru_spec* and the paths that are affected.

Syntax

path_delay_from_to_thru_spec_0 ::= **(** *label?* **PATH_DELAY**
path_delay_value from_to_thru_spec+ **)**

Example

```
(PATH_DELAY 3.0 4.0 2.5 2.9
  (PATH
    (FROM and1.a)
    (THRU_ALL and2.a and3.a)
    (TO and4.a)
  )
)
```

This example specifies the combinational delay for all paths that start at *and1.a*, go through *and2.a* then *and3.a*, then end at *and4.a*.

The delay through each path of the transitions resulting in a rising transition at *and4.a* must be greater than 3.0 ns and less than 4.0 ns.

The delay through each path of the transitions resulting in a falling transition at *and4.a* must be greater than 2.5 ns and less than 2.9 ns.

Slew Limit

The **SLEW_LIMIT** construct is the preferred way to specify a constraint on edge transition time as measured at a specified port (input, output or bidirectional). The related MAX_TRANSITION_TIME construct is archaic.

Syntax

slew_limit_spec ::= **(** *label?* **SLEW_LIMIT** *slew_value* *port_instance_or_master** **)**

The *slew_value* is a time value and must be specified in the units defined by the *time_scale*. It follows the convention for *rise_fall_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

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.

Each *port_instance* must be a port on a cell contained within the current GCF cell. The default slew limit, which can be set by omitting the *port_instances*, applies to all ports on all cells contained within the current GCF cell.

A master-based default slew limit can also be specified using *port_master*. The master-based default slew limit applies to all occurrences of the *scalar_port* on cell instances of type *cell_id*.

Precedence Rules

- Usually, the slew limit is specified in the library. If the slew limit is specified in both the library and the GCF file, the more restrictive constraint will be used.
- Explicit slew limits have higher precedence than master-based default slew limits, which have higher precedence than normal default slew limits.
- The **SLEW_LIMIT** construct and the **MAX_TRANSITION_TIME** construct (archaic) have the same precedence for the maximum slew limit. If both are specified in the same GCF file, the last one given will be used.

Example

```
(SLEW_LIMIT 2.0 3.0 2.5 3.5 out1)
```

This example specifies that the slew (transition time) at *out1* must be between 2.0 and 3.0 ns for the rising transition, and between 2.5 and 3.5 ns for the falling transition.

Example

```
(SLEW_LIMIT * 1.5 * 1.8
  ((CELLTYPE dff) d)
)
```

This example constrains the maximum slew (transition time) at the *d* input of all instances of the *dff* cell type within the current GCF cell and its descendents. The slew must be less than 1.5 ns for the rising transition, and less than 1.8 ns for the falling transition.

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 data 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. The *borrow_value* applies for all operating points.

Syntax

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

borrow_value ::= NUMBER

borrow_item ::= *port_instance*
||= *cell_instance*
||= *waveform_name*
||= *typed_waveform_list*

If no *borrow_items* are specified, borrowing will be restricted for all level-sensitive latches within the current GCF cell and its descendents.

If a *port_instance* that was identified as a clock (through the **CLOCK** construct—see “Clock Specifications” on page 85) is specified, borrowing will be restricted for all data inputs of all level-sensitive latches in the transitive fanout of that clock. Otherwise, the *port_instances* must be data input pins or clock input pins of level-sensitive latches. When a clock input is specified, it affects the borrowing for all of the related data inputs on the latch.

If a *cell_instance* is specified, it must be a level-sensitive latch, and borrowing is restricted on all data inputs of the latch.

If a *waveform_name* is specified, borrowing will be restricted for all data inputs on level-sensitive latches in the transitive fanout of the clocks associated with that waveform.

When several borrow limits specified in different ways affect the same data input, the tightest limit is used.

Example

```
(BORROW_LIMIT 3.0 latch1.clk)
```

This example constrains borrowing for all data inputs on latch1 related to clk to use no more than 3.0 ns of the available portion of the pulse width.

Clock Mode

The **CLOCK_MODE** construct is used to specify the default clock mode, which affects computation of

- the insertion delay between *clock_roots* specified in **CLOCK** constructs) and primitive clock input pins

- the slew at primitive clock input pins.

Syntax

clock_mode_spec ::= **(** *label?* **CLOCK_MODE** *clock_mode_value* **)**

clock_mode_value ::= **IDEAL**
||= **ACTUAL**

The default clock mode for the current GCF cell and its descendents can be explicitly specified using the **CLOCK_MODE** construct. When the clock mode is specified at several levels in the design hierarchy, the mode specified at the lowest level has precedence. If no clock mode is specified, the default clock mode is **IDEAL**.

The default clock mode applies to all clock paths starting at *clock_roots* within the current GCF cell, including the primary inputs of the cell. The default clock mode can be overridden for particular clock networks by specifying the clock mode explicitly in a **CLOCK_DELAY** construct.

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

The **CLOCK_DELAY** constraints describe the worst case characteristics of the network, and they are often used in constructing the final version of the network.

In addition, the constraints are used for worst case analysis of the design before the final network is created. In the early stages of the design flow, the design will often contain a preliminary network. To ensure that the design will still perform correctly once the final network is created, analysis tools may ignore the preliminary network, using the worst case characteristics from the **CLOCK_DELAY** constraints instead.

CLOCK_DELAY Scope

The scope of the clock distribution network includes the root, the leaf pins, and the cells and nets between the root and the leaf pins. Leaf pins lie on the boundary of the clock network; no logic beyond a leaf pin is included in the clock network.

Each leaf pin must be reachable by tracing forward from the root through interconnect, buffers, inverters, and possibly combinational logic gates.

An error will be reported if the leaf pin is reachable by tracing forward through any non-unate timing arcs.

Generally there will only be one path between the root and a given leaf pin, but in certain cases (such as parallel buffers driving an internal net within a clock tree, or a clock mesh) there may be several paths. An error will be reported if two paths through the clock distribution network to a given leaf pin to have different unateness (either positive unate or negative unate are allowed, but not both).

Leaf pins fall into three categories:

- Default leaf pins. These are all of the clock input pins on primitives that are reachable from the root without tracing through any explicit leaf pins. In Figure 10, the clock inputs on FF1, FF2, FF3, and FF4 are default leaf pins.

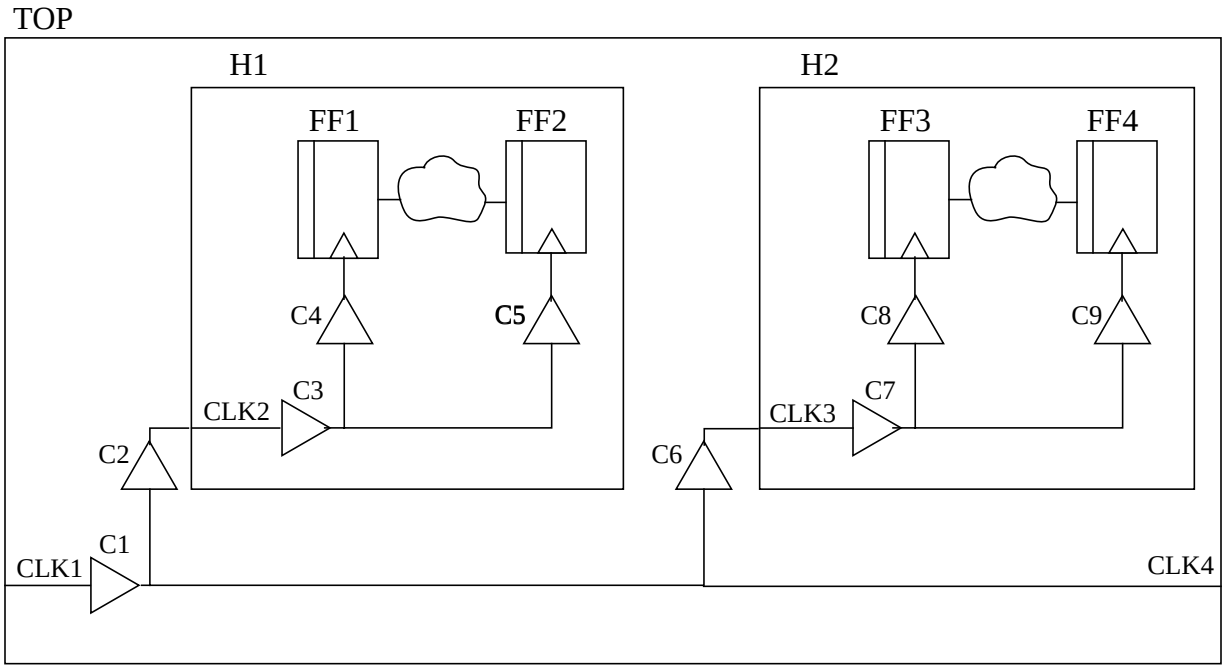
  Default leaf pins must be identified in the library as clock pins. The relevant edges of the clock at a default leaf pin are determined either explicitly from additional attributes on the pin, or implicitly from the timing checks and delay arcs related to the pin. When specified, the additional attributes indicate whether the pin is rising or falling edge-triggered, or active low or high level-sensitive.

- Explicit leaf pins. These are port instances (pins on primitives or primary output/bidirectional pins on the cell) that are explicitly described in the **CLOCK_DELAY** construct. In Figure 10, *CLK4* must be specified as an explicit leaf pin.

  Normally, the logic beyond an explicit leaf pin is part of a larger clock distribution network, or the leaf pin is a normal clock input pin on a register. In certain cases, the circuit beyond the leaf pin uses the clock signal as if it were a data signal. The *data_leaf* construct should be used to identify these cases.

When modeling hierarchical clock trees, each GCF must only specify a **CLOCK_DELAY** construct for the highest *clock_delay_root* contained within the portion of the design described by the GCF. An error message will be given if a *clock_delay_root* for a **CLOCK_DELAY** construct lies in the transitive fanout of a *clock_delay_root* for another **CLOCK_DELAY** construct.

Figure 10 Hierarchical Clock Tree



For example, in the circuit shown in Figure 10, the GCF used for doing analysis on the *H1* level of hierarchy must specify *CLK2* as the *clock_delay_root*, while the GCF used for doing analysis on the *TOP* level of hierarchy must specify *CLK1* as the *clock_delay_root*.

Insertion delays specified in the **CLOCK_DELAY** construct describe the insertion delay from the *clock_delay_root* all the way to the primitive leaf pins, even when the clock tree is constructed hierarchically.

In the circuit shown in Figure 10, the **CLOCK_DELAY** construct in the GCF used for doing analysis on the *H1* level of hierarchy should describe the nominal insertion delay from *CLK2* to the clock input pins on *FF1* and *FF2*. The **CLOCK_DELAY** construct in the GCF used for doing analysis on the *TOP* level of hierarchy should describe the nominal insertion delay from *CLK1* to the clock input pins on *FF1*, *FF2*, *FF3*, and *FF4*.

Syntax

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

clock_delay_root ::= *untyped_port_instance*
||= **(** *cell_instance input_port output_port* **)**
||= *waveform_name*

If a *port_instance* is specified for the *clock_delay_root*, it indicates the pin that is the source of the 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_delay_root*, it gives the instance name of a cell that drives the network. Insertion delay is measured from the specified *input_port* through the *output_port* to each of the leaf *port_instances*.

If a *waveform_name* is specified for a *clock_delay_root*, it affects both internal and external clock networks associated with the waveform. When a *waveform_name* is used, no leaf *port_instances* may be specified.

Internal clock networks are contained within the current GCF cell. When a *waveform_name* is specified for the *clock_delay_root*, the **CLOCK_DELAY** construct defines default values for all of the internal clock networks that are associated with that waveform. Insertion delay, skew, and slew may be specified for internal clock networks.

The **CLOCK** construct is used to associate a *waveform_name* with the *port_instance* that is the root of an internal clock network. When a *waveform_name* is used for the *clock_delay_root*, the corresponding **CLOCK** construct must precede the **CLOCK_DELAY** construct in the GCF file.

The default values specified by a **CLOCK_DELAY** with a *waveform_name clock_delay_root* can be overridden for a particular internal clock network by another **CLOCK_DELAY** construct that explicitly specifies the root *port_instance* or *cell_instance*.

External clock networks are not contained within the current GCF cell. The **CLOCK_DELAY** construct for an external clock network must only specify insertion delay. Insertion delay on an external clock network affects the effective offset of the reference waveform edge for **ARRIVAL** and **REQUIRED** constructs.

Skew within an external clock network, or between an external clock network and internal clock networks, must be specified using the **CLOCK_UNCERTAINTY** construct.

Slew is not relevant for external clock networks, because the leaf pins for an external clock network are not visible within the current GCF cell.

Syntax

leaf_spec ::= *clock_mode_value*
||= *default_leaf_spec*
||= *explicit_leaf_spec*

default_leaf_spec ::= **(** *default_leaf_option+* **)**

default_leaf_option ::= *insertion_delay_spec*
||= *clock_skew_spec*
||= *clock_slew_spec*

explicit_leaf_spec ::= **(** *explicit_leaf_option* clock_delay_leaf+* **)**

explicit_leaf_option ::= *insertion_delay_spec*
||= *internal_insertion_delay_spec*
||= *clock_slew_spec*

clock_delay_leaf ::= *clock_leaf*
||= *data_leaf*

data_leaf ::= **(DATA** *port_instance+* **)**

insertion_delay_spec ::= **(INSERTION_DELAY**
insertion_delay_value **)**

internal_insertion_delay_spec ::= **(INTERNAL_INSERTION_DELAY**
insertion_delay_value **)**

clock_skew_spec ::= **(SKEW** *skew_value* **)**

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

insertion_delay_value ::= *rise_fall_min_max*
skew_value ::= *rise_fall_min_max*
slew_value ::= *rise_fall_min_max*

The *default_leaf_spec* describes the default nominal insertion delay, the default nominal slew at the leaf pins, and the nominal skew for the network as a whole.

An *explicit_leaf_spec* overrides the *default_leaf_spec* values for particular *clock_leafs*. In particular, an *explicit_leaf_spec* can override the insertion delay to a *clock_leaf* and the nominal slew at the *clock_leaf*. When an *explicit_leaf_spec* is used to override the insertion delay to a *clock_leaf*, that *clock_leaf* is not included in skew computations for the network as a whole.

When specified in a **CLOCK_DELAY** construct for a particular internal clock network, the *clock_mode* overrides any default clock mode specified using the **CLOCK_MODE** construct. An error message will be given if a *clock_mode* is specified for an external clock network. Since the logic in the external clock network is not visible within the current design, the analysis of external clock networks is always **IDEAL**.

The *data_leaf* form is used when an output from the clock network is used as a data signal. Normally, the logic beyond an explicit leaf pin is part of a larger clock distribution network, or the leaf pin is a normal clock input pin

on a register. In certain cases, the circuit beyond the leaf pin uses the clock signal as if it were a data signal. The *data_leaf* construct should be used to identify these cases. Leaf pins listed in the *data_leaf* construct are part of the clock network and are treated like any other explicit leaf pins; only the analysis of the logic beyond the leaf pin is affected.

The *internal_insertion_delay_spec* can be used when a *clock_leaf* is an input on a hierarchical block and a timing model is used for the block. Generally the timing model should describe the internal insertion delay within the hierarchical block, from the input port to the network leaf pins within the block. When the timing model does not include the internal insertion delay, it can be specified in the GCF instead. Values specified in the GCF override values in the timing model.

The default insertion delay and any explicit insertion delays represent the complete insertion delay from the *clock_delay_root* to all of the primitive leaf pins, even when those primitive leaf pins are contained within a hierarchical block and a timing model is used for that block.

Therefore, the partial insertion delay from the *clock_delay_root* to the input of a hierarchical block is the difference between the complete insertion delay and the internal insertion delay within the hierarchical block. When the input of a hierarchical block is specified as a *clock_leaf* in an *explicit_leaf_spec*, the insertion delay still represents the complete insertion delay, not the partial insertion delay up to that input.

insertion_delay_value, *skew_value*, and *slew_value* are time values and must be specified in the units defined by the *time_scale*. They follow the convention for *rise_fall_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

As a special case, it is legal to specify place-holders for all of the *insertion_delay_value* fields or all of the *slew_value* fields in an *explicit_leaf_spec*. When place-holders are used for all of the fields, it indicates that the insertion delay or slew is unconstrained for that *clock_leaf*; the *default_leaf_spec* does not apply. Ordinarily, if an **INSERTION_DELAY** construct is specified in a *default_leaf_spec*, it applies to all primitive leaf pins, including *clock_leafs* that are listed in *explicit_leaf_specs* that do not specify an **INSERTION_DELAY**.

The rise fields in *insertion_delay_value*, *skew_value*, and *slew_value* correspond to the rising transition of the clock at the primitive leaf pins. The rise fields in *skew_value* represent the nominal skew between the

rising transitions at any pair of primitive leaf pins. Similarly, the fall fields correspond to the falling transition at the primitive leaf pins.

In GCF 1.4, only a single operating point can be modeled with the **OPERATING_CONDITIONS** construct. This leads to ambiguities because *insertion_delay_value*, *skew_value*, and *slew_value* all support minimum and maximum fields for historical reasons that are no longer valid.

A future version of GCF is expected to support multiple operating points, and at that time, the minimum fields will correspond to best case operating conditions while the maximum fields will correspond to worst case operating conditions.

For GCF 1.4, in general

- The mininimum and maximum fields in *insertion_delay_value* should both be set to the same value:
  - The nominal insertion delay expected at the operating point specified in the **OPERATING_CONDITIONS** construct.

  The minimum insertion delay will be used for the source clock delay in hold checks, and for the target clock delay in setup checks.

  The maximum insertion delay will be used for the source clock delay in setup checks, and for the target clock delay in hold checks.
- The minimum and maximum fields in *skew_value* should both be set to the same value:
  - The largest skew expected between any pair of *clock_leafs* at the operating point specified in the **OPERATING_CONDITIONS** construct.

  The minimum skew will be added to the target clock delay for hold checks.

  The maximum skew will be subtracted from the target clock delay for setup checks.
- The minimum and maximum fields in *slew_value* should both be set to the same value:
  - The nominal slew expected at the operating point specified in the **OPERATING_CONDITIONS** construct.

  The minimum slew will be used in calculating minimum delays downstream of the *clock_leafs*.

  The maximum slew will be used in calculating maximum delays downstream of the *clock_leafs*.

Often only one transition of the signal is relevant for a particular *clock_leaf*, or for all of the leaf pins in the network. In that case, asterisks should be used as place-holders for the values associated with the other transition.

For example, all of the leaf pins of a clock network may be clock inputs on rising edge-triggered flip flops. In that case, asterisks should be used for the fall min and max entries in the *insertion_delay_value*, *skew_value*, and *slew_value*.

If the *default_leaf_spec* gives values for both the rising and falling transitions, only the values for the relevant transition are used for each default leaf pin.

For an explicit leaf pin, the relevant edges are determined by the values given in the *explicit_leaf_spec* and the *default_leaf_spec*. If place-holders are given for an edge in the *explicit_leaf_spec*, then that edge is treated as not relevant for the leaf pin, even if values are given for both edges in the *default_leaf_spec*.

When the *clock_delay_root* is a *port_instance* or a *cell_instance*, the *skew_value* is treated as an uncertainty that only applies when analyzing paths between registers within the same clock network. This uncertainty has higher precedence than target-based uncertainty or inter-clock uncertainty for those paths (see “Inter-Clock Uncertainty” on page 151).

When the *clock_delay_root* is a *waveform_name*, the *skew_value* treated as an uncertainty that only applies when analyzing paths between registers within the same clock network, for each of the clock networks that distribute that waveform. The *skew_value* does not apply for paths between registers in different clock networks, even if both clock networks distribute the same waveform. In addition, the skew_value does not apply to any external clock networks implied by **ARRIVAL** or **REQUIRED** constructs that reference the *waveform_name*.

Precedence Rules

When the analysis mode for the network driving a leaf pin is ideal, the slew values at the leaf pin will be determined using the following precedence order:

- The slew specified explicitly by an **INTERNAL_SLEW** construct
- The slew specified in an *explicit_leaf_spec* for the leaf pin
- The slew specified in a *default_leaf_spec* for a **CLOCK_DELAY** construct that includes the pin as an implicit leaf pin
- The default **INTERNAL_SLEW**

- The default **INPUT_SLEW**
- 0

When the analysis mode for the network driving a leaf pin is to use actual delays, the slew values at the leaf pin will be determined using the following precedence order:

- The slew specified explicitly by an **INTERNAL_SLEW** construct
- The calculated slew

Example

```
(CLOCK_DELAY

  // root
  clk1

  // defaults (apply to all leaf pins)
  (
    (INSERTION_DELAY 5.0 6.0 * *)
    (SKEW 1.0 1.3 * *)
    (SLEW 0.5 0.7 * *)
  )
  // explicit leaf pins, rising edge, default values
  (clk_out)

  // explicit leaf pins, both edges active
  (
    (INSERTION_DELAY 5.0 6.0 4.0 5.0)
    (SLEW 0.5 0.7 0.3 0.4)
    a/dsp/cp
  )

  // explicit leaf pins, rising edge, overriding default
  (
    (INSERTION_DELAY 4.0 4.5 * *)
    a/ff3/cp
    a/ff4/cp
  )

  // data leaf pin, both edges active
  (
    (INSERTION_DELAY 5.0 6.0 4.0 5.0)
    (SLEW 0.5 0.7 0.3 0.4)
    clock_active/a
  )


) // clock_delay
```

This example specifies a complicated clock network with a primary input port instance as a root and a mixture of default, explicit, and data leaf pins.

The default insertion delay for the rising edge at the leaf pins is 5.0 ns at the minimum operating point, and 6.0 ns at the maximum operating point.

The explicit leaf pins are *clk_out*, *a/dsp/cp*, *a/ff3/cp*, *a/ff4/cp*. In addition, *clock_active/a* is a data leaf, where logic beyond that pin is treated as data logic rather than clock logic.

Inter-Clock Uncertainty

In general, there is always some skew between when the launching clock edge arrives at a source register and when the capturing clock edge arrives at a target register.

If both the source register and the target register are contained within the portion of the design described by the GCF, and they are both driven by the same clock network, then the nominal skew can be described directly using the **CLOCK_DELAY** construct. Once the real clock network has been added to the design, the actual skew can be computed by analyzing just the portion of the clock network that is visible within the current GCF cell.

If the source and target clocks are derived from a common oscillator, but only a portion of the clock network relating the clocks is visible within the current GCF cell, then the skew between the clocks is affected by insertion delays both internal and external to the current GCF cell.

External insertion delays can be described with the **CLOCK_ARRIVAL** construct. The nominal internal insertion delays can be described with the **CLOCK_DELAY** construct. Once the real clock network has been added to the design, the actual internal insertion delays can be computed.

However, when the design is partially complete both the external and internal insertion delays may not be known. The designer may expect to be able to balance the insertion delays between different clock sub-trees as well as minimize skew within each clock sub-tree. In this case, the expected difference between the insertion delays may be known, but the insertion delays themselves may not.

The designer may also want to add some margin in the analysis to account for any incremental changes that may be necessary, or specify that there is some uncertainty associated with the insertion delays specified in the **CLOCK_ARRIVAL** and **CLOCK_DELAY** constructs.

The **CLOCK_UNCERTAINTY** construct is used to specify these types of skew, uncertainty, and margin.

Syntax

clock_uncertainty_spec ::= **(** *label?* **CLOCK_UNCERTAINTY**
*clock_uc_option**
clock_uc_value
clock_uc_item **)**

clock_uc_option ::= *clock_uc_calc_option*
||= *clock_uc_mode_option*

clock_uc_calc_option ::= **ABSOLUTE**
||= **INCREMENT**

clock_uc_mode_option ::= **IDEAL**
||= **ACTUAL**

clock_uc_value ::= *r_min_max*

clock_uc_item ::= *target_clock_uc_item+*
||= *target_clock_uc_item_edge*
||= *inter_clock_uc_item*

target_clock_uc_item ::= *waveform_name*
||= *typed_waveform_list*
||= *clock_root*
||= *clock_leaf*
||= *clock_leaf_instance*

clock_leaf_instance ::= *cell_instance*

target_clock_uc_item_edge ::= **(** *waveform_edge target_clock_uc_item+* **)**

inter_clock_uc_item ::= **(BETWEEN**
inter_clock_from
inter_clock_to **)**

inter_clock_from ::= **(FROM** *inter_clock_from_to_item* **)**

inter_clock_to ::= **(TO** *inter_clock_from_to_item* **)**

inter_clock_from_to_item ::= *waveform_name*
||= *waveform_edge*

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

The *clock_uc_value* follows the conventions for *r_min_max* described in “Value Types” on page 48, as well as the semantics for operating points described in “Min/Max Values and Operating Conditions” on page 51.

In GCF 1.4, only a single operating point can be modeled with the **OPERATING_CONDITIONS** construct. This leads to ambiguities because *clock_uc_value* supports both minimum and maximum fields, for compatibility with a future version of GCF that is expected to support multiple operating points. At that time, the minimum fields will correspond to best case operating conditions while the maximum fields will correspond to worst case operating conditions.

For GCF 1.4, in general the mininimum and maximum fields in *clock_uc_value* should both be set to the same value:

- The largest uncertainty expected at the operating point specified in the **OPERATING_CONDITIONS** construct.

The minimum uncertainty will be added to the target clock delay for hold checks.

The maximum uncertainty will be subtracted from the target clock delay for setup checks.

The uncertainty described by the *clock_uc_value* can either override or add to any skew that is computed from the insertion delays to the source and target registers.

- When the **ABSOLUTE** keyword is specified, the *clock_uc_value* overrides computed skew, and the insertion delays to the source and target registers are ignored.
- When the **INCREMENT** keyword is specified, the *clock_uc_value* is added to the skew computed from the insertion delays.
- If neither option is specified, the default is **INCREMENT**.

The *clock_uc_mode_option* is used to specify different uncertainty values to be used based on the analysis mode for the source and target clock networks:

- The uncertainty value specified in a **CLOCK_UNCERTAINTY** construct with the **IDEAL** *clock_uc_mode_option* is used when the analysis mode for either the source or the target clock network is ideal.
- The uncertainty value specified in a **CLOCK_UNCERTAINTY** construct with the **ACTUAL** *clock_uc_mode_option* is used when the actual delays are used for both the source and target clock networks.

If no *clock_uc_mode_option* is specified, the uncertainty value applies to both modes.

Target-Based Uncertainty

Target-based uncertainty is specified using the *target_clock_uc_item* and *target_clock_uc_item_edge* forms. Target-based uncertainty affects all paths where data is captured by the clock edges referenced by the *target_clock_uc_item* or *target_clock_uc_item_edge*.

For the *target_clock_uc_item* form, the uncertainty affects both rising and falling capturing edges. For the *target_clock_uc_item_edge* form, the uncertainty only affects the specified type of capturing edge.

When a *waveform_name* is specified, the uncertainty value affects paths to registers in the transitive fanout of the *clock_roots* associated with the waveform. It also affects paths to the *port_instances* that have a **REQUIRED** time referenced to the waveform.

When a *clock_root* is specified for *target_clock_uc_item*, the uncertainty value affects paths to all of the data inputs of registers in the transitive fanout of the *clock_root*.

When a clock input pin of a register is specified using the *clock_leaf* form of *target_clock_uc_item*, the uncertainty value affects paths to the related data inputs of that register.

When a register is specified using the *clock_leaf_instance* form of *target_clock_uc_item*, the uncertainty value affects paths to all of the data inputs of that register.

Inter-Clock Uncertainty

Inter-clock uncertainty is specified using the *inter_clock_uc_item* form. The uncertainty affects paths between

- Source registers in the transitive fanout of the *clock_roots* associated with the *inter_clock_source* waveform
- Target registers in the transitive fanout of the *clock_roots* associated with the *inter_clock_target* waveform.

Inter-clock uncertainty also affects paths from *port_instances* that have an **ARRIVAL** time referenced to the *inter_clock_source* waveform, and paths to *port_instances* that have a **REQUIRED** time referenced to the *inter_clock_target* waveform.

When a *waveform_edge* is specified for the *inter_clock_source*, only the paths that are launched from that edge are affected. When a *waveform_edge* is specified for the *inter_clock_target*, only the paths that are captured by that edge are affected.

Precedence Rules

- Intra-tree uncertainty specified in the **CLOCK_DELAY** construct always has the highest precedence for paths between source and target registers that are both driven by that tree.
- Edge-specific uncertainty has higher precedence than non-edge-specific uncertainty. For inter-clock skew, **TO** edge specifications have higher precedence than **FROM** edge specifications.
- Uncertainties specified using the **CLOCK_DELAY SKEW** construct or the **CLOCK_UNCERTAINTY** construct are added to uncertainty due to jitter specified using **WAVEFORM** or **DERIVED_WAVEFORM**.

- The precedence between different target-based uncertainties of the same edge type is (in decreasing order): *clock_leaf*, *clock_root*, *waveform_name*.

Example

```
(CLOCK_UNCERTAINTY
  ABSOLUTE
  IDEAL
  1.0 1.3
  "wave"
)
(CLOCK_UNCERTAINTY
  INCREMENT
  ACTUAL
  0.1
  "wave"
)
```

In this example, different target-based clock uncertainties are specified for **IDEAL** mode and **ACTUAL** mode. When ideal insertion delays are used for either the source or target clock network, the first clock uncertainty overrides the skew computed from the insertion delays of the clock networks.

- 1.3 ns of uncertainty is subtracted from the target clock edge for setup checks.
- 1.0 ns of uncertainty is added to the target clock edge for hold checks.

When actual insertion delays are used for both the source and target clocks, a margin of 0.1 ns is added to the skew computed from the actual insertion delays:

- 0.1 ns of uncertainty is subtracted from the target clock edge for setup checks
- 0.1 ns of uncertainty is added to the target clock edge for hold checks.

Example

```
(CLOCK_UNCERTAINTY
  * 1.3
  clk2
)
(CLOCK_UNCERTAINTY
  * 0.7
  (BETWEEN
    (FROM (posedge "wave1"))
    (TO (negedge "wave2"))
  )
)
```

In this example, both a target-based and an inter-clock uncertainty are specified. The target-based clock uncertainty affects setup checks on paths to registers in the transitive fanout of the clock root *clk2*. The inter-clock uncertainty affects setup checks on paths launched by the rising edge of waveform *wave1* and captured by the falling edge of waveform *wave2*.

Assuming that *wave2* is associated with *clk2*, the inter-clock uncertainty has higher precedence for target registers in the transitive fanout of *clk2* that have a falling capturing edge. Therefore, for setup checks on paths from a flip flop triggered by the rising edge of *wave1* to a falling edge-triggered flip flop in the transitive fanout of *clk2*, an uncertainty of 0.7 will be subtracted from the target clock edge.

For setup checks at a rising edge-triggered flip flop in the transitive fanout of clk2, an uncertainty of 1.3 will be subtracted from the target clock edge The same will be true for setup checks at falling edge-triggered flip flops in the transitive fanout of *clk2* where the source register is not triggered by the rising edge of *wave1*.

The constructs in this example do not affect the uncertainty used for hold checks.

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.

Archaic Constructs

Archaic Timing Exception Constructs

This section describes archaic constructs, which are supported in this version of GCF for backward compatibility, but may be dropped in the next major version.

In general, tools are expected to be able to read any version of GCF starting with 1.0, including the following constructs when used in a GCF file with a **VERSION** string containing 1.4 and lower. However, the following constructs may not be supported when used in a GCF file with a **VERSION** string containing 2.0 or higher.

Level 1 Port Instance Edge Specification (Archaic)

The Level 1 thru_edge_spec construct is archaic and has been replaced by *from_to_thru_path_spec*. The thru_edge_spec construct constrains all paths that pass through a given *port_instance*, and it affects only the transitions through those paths that result in the given edge at that *port_instance*.

The semantics of which paths are constrained when the *port_instance* is on a flip flop or a latch are the same as for the Level 0 *thru_spec* construct, except that only the specified edges of those paths are constrained.

Syntax

thru_edge_spec ::= (**THRU** port_instance_edge) (archaic)

Example

```
(THRU (negedge ff1.SN))
```

This example constrains all paths through the preset input of a flip flop. Only the transitions through those paths that result in a falling transition at the preset input are affected.

Level 1 Arc Edges Specifications (Archaic)

The Level 1 arc_edges_spec is archaic and has been replaced by *from_to_thru_path_spec*. The arc_edges_spec constrains certain edges of all paths that pass through two *port_instances*, including paths that 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).

Syntax

arc_edges_spec ::= (**ARC**
port_instance_edge
port_instance_edge) (archaic)

Example

```
(ARC (posedge DRVR_A.Z) (posedge RCVR_A.A))
```

This example constrains all paths that pass through the output of a tristate bus driver, DRVR_A.Z, then back through the input of a tristate bus receiver, RCVR_A.A. Only the rising edge at the two *port_instances* is affected; the falling edge at the two *port_instances* is not affected.

Level 1 Thru All Specification (Archaic)

The Level 1 thru_all_spec construct is archaic and has been replaced by *from_to_thru_path_spec*. The thru_all_spec constrains all paths that pass through all of the listed *port_instances*. 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.

Syntax

```
thru_all_spec ::= ( THRU_ALL
                    port_instance
                    port_instance+ )          (archaic)
```

Example

```
(THRU_ALL
  IN1
  X.A
  Y.A
)
```

This example constrains all paths that start at a primary input (IN1) then pass through *port_instances* X.A and Y.A. All transitions through these paths are affected.

Level 1 Thru All Edges Specification (Archaic)

The Level 1 thru_all_edges_spec construct is archaic and has been replaced by *from_to_thru_path_spec*. The thru_all_edges_spec constrains paths that go through all of a set of *port_instances*, which do not have to be contiguous in the paths. Only the transitions through the constrained paths that result in the specified edge at each of the *port_instances* are affected.

Syntax

```
thru_all_edges_spec ::= ( THRU_ALL
                          port_instance_edge
                          port_instance_edge+ )   (archaic)
```

Example

```
(THRU_ALL
  (posedge IN1)
  (posedge X.A)
  (negedge Y.A)
)
```

Archaic Constructs

This example constrains all paths that start at a primary input (IN1) then pass through *port_instances* X.A and Y.A. Only the transitions through the path that result in a rising edge at IN1 and X.A and a falling edge at Y.A are affected.

Disabling Paths Through Edges (Archaic)

The disable_edges_spec_1 construct construct is archaic and has been replaced by *disable_from_to_thru_spec_0*. 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 late (maximum) or the early (minimum) timing checks must be disabled; otherwise, both the early and late timing checks are disabled.

Syntax

```
disable_edges_spec_1  ::=  ( label? DISABLE
                                 disable_edges_path_spec+
                                 timing_check? )                (archaic)
disable_edges_path_spec::= thru_edge_spec
                       ||= arc_edges_spec
                       ||= thru_all_edges_spec                  (archaic)
```

See pages 158-159 for details on the types of paths that can be constrained by the *thru_edge_spec*, the *arc_edges_spec*, and the *thru_all_edges_spec*.

For each of these cases, if the **HOLD** or **SETUP** keyword is specified as the *timing_check*, only the early (minimum) or the late (maximum) timing checks must be disabled. Otherwise, both the early and late timing checks are disabled.

Level 1 Multi-Cycle Paths

In Level 1, the constrained paths can be specified in additional ways. The multi_cycle_thru_spec_1 construct is archaic and has been replaced by the *multi_cycle_from_to_thru_spec_0* construct.

Syntax

```
multi_cycle_spec_1  ::=  multi_cycle_thru_spec_1      (archaic)
```

Multi-Cycle Paths With Arc and Thru (archaic)

The multi_cycle_thru_spec_1 construct is archaic and has been replaced by the *multi_cycle_from_to_thru_spec_0* construct.

Syntax

```
multi_cycle_thru_spec_1  ::=  ( label? MULTI_CYCLE
                                  multi_cycle_option+
                                  multi_cycle_thru_path_spec_1+ )
                                      (archaic)
multi_cycle_thru_path_spec_1  ::=  arc_spec
                              ||=  thru_spec
                              ||=  thru_all_spec                (archaic)
```

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.

Archaic Constructs

Level 1 Path Delays

In Level 1, the constrained paths can be specified in additional ways.

Syntax

```
path_delay_spec_1  ::=  path_delay_path_spec_1      (archaic)
```

The path_delay_path_spec_1 construct is archaic and has been replaced by the *path_delay_from_to_thru_spec_0* construct.

Syntax

```
path_delay_path_spec_1 ::=  ( label? PATH_DELAY
                                path_delay_value
                                path_delay_path_spec_1+ )      (archaic)
path_delay_path_spec_1 ::=  arc_spec
                       ||=  thru_spec
                       ||=  thru_all_spec                      (archaic)
```

Example

```
(PATH_DELAY 3.0 * 2.7 *
  (ARC ff1.clk ff1.q)
)

(PATH_DELAY 4.5 * 4.3 *
  (ARC ff1.clk ff1.qn)
)
```

This example specifies the combinational delay for all paths that start at *ff1.clk* and go through either *ff1.q* or *ff1.qn*. The constraints are different based on the output of the flip flop.

For each path that goes through the *q* output, the delay of the transitions that result in a rising transition at the target (which will be either a register data input or a primary output) must be greater than 3.0 ns at the minimum operating point. The delay for a falling transition at the target must be greater than 2.7 ns. The late (maximum) delays are unconstrained.

For each path that goes through the *qn* output, the delay of the transitions that result in a rising transition at the target (which will be either a register data input or a primary output) must be greater than 4.5 ns at the minimum operating point. The delay for a falling transition at the target must be greater than 4.3 ns. The late (maximum) delays are unconstrained.

Max Transition Time

The MAX_TRANSITION_TIME construct is archaic. It only allows specifying a maximum constraint value. The preferred construct, **SLEW_LIMIT**, allows both a maximum and a minimum.

Syntax

```
max_transition_time_spec  ::=  ( label? MAX_TRANSITION_TIME
                                    rise_fall
                                    port_instance* )              (archaic)
```

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.

The *rise_fall* parameter is a time value and it follows the same conventions for units and thresholds as the *slew_value* in **SLEW_LIMIT**. The same values apply for all operating points.

Example

```
(MAX_TRANSITION_TIME 1.5 1.8 ff1.d)
```

This example constrains the maximum slew (transition time) at *ff1.d*. The slew must be less than 1.5 ns for the rising transition, and less than 1.8 ns for the falling transition.

Archaic Constructs

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 41, “Meta Data” on page 44, and “Include Files” on page 46.

Syntax

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

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

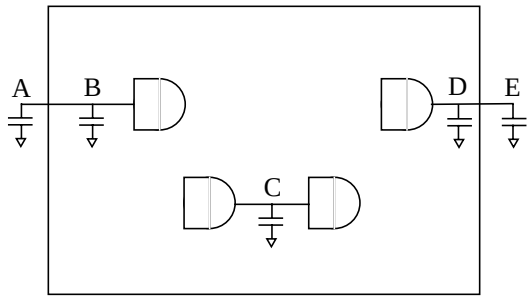
parasitics_subset_spec ::= *parasitics_environment*
||= *parasitics_constraints*
||= *extension*
||= *meta_data*

Example

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

Figure 11 below summarizes the different types of parasitic environment and constraint specifications.

Figure 11 Parasitics Environment and Constraints



A represents the external load on an input interface net of the current GCF cell, which is an environment condition specified using the **EXTERNAL_LOAD** construct. The external load affects the delay calculation on the interface net.

B represents the internal load on an input interface net of the current GCF cell, which is a constraint specified using the **INTERNAL_LOAD** construct. The internal load constraint affects optimization tools, which will try to ensure that the actual load within the boundaries of the current GCF cell meets the constraint.

C represents the load on a net which is entirely contained within the current GCF cell, which is a constraint specified using the **LOAD** construct. The load constraint affects optimization tools, which will try to ensure that the actual load meets the constraint.

D represents the internal load on an output interface net of the current GCF cell, which is a constraint specified using the **INTERNAL_LOAD** construct. The internal load constraint affects optimization tools, which will try to ensure that the actual load within the boundaries of the current GCF cell meets the constraint.

E represents the external load on an output interface net of the current GCF cell, which is an environment condition specified using the **EXTERNAL_LOAD** construct. The external load affects the delay calculation on the interface net.

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*

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_wire_load_model_spec*
::= *wire_load_model_spec*
||= *meta_data_1*

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

External Loading

For an interface net that is connected to a primary port on the current GCF cell, the **EXTERNAL_LOAD** construct specifies the actual value of the portion of the capacitance which is not contained within the current GCF cell, including the pin capacitance of any pins connected to the net outside of the current GCF cell.

INTERNAL_LOAD specifies the capacitance allowed inside the current GCF cell, while **EXTERNAL_LOAD** specifies the capacitance that exists outside the current GCF cell.

Syntax

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

capacitance_value ::= *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. The *capacitance* value follows the conventions for *min_max* described in “Value Types” on page 48.

The external load is added to the capacitance within the cell when computing the delay of the interface net.

External Fanout

For an interface net that is connected to a primary port on the current GCF cell, the **EXTERNAL_FANOUT** construct specifies the number of ports connected to the net outside of the current GCF cell. This, combined with a wire load model named in the **EXTERNAL_WIRE_LOAD_MODEL** construct, specifies the portion of the capacitance on the interface net which is not contained within the current GCF cell.

INTERNAL_FANOUT specifies the number of ports that are allowed to be connected to the net inside the current GCF cell. **EXTERNAL_FANOUT** specifies the number of ports which are connected to the net outside the current GCF cell.

This construct is a Level 1 construct because it requires a separate source of wire load models for proper interpretation. The wire load models are not defined in GCF, simply referenced by name.

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. The *num_loads* value follows the conventions for *min_max* described in “Value Types” on page 48.

The external load computed from the external fanout is added to the capacitance within the cell when computing the delay of the interface net

External Wire Load Model

.For an interface net that is connected to a primary port on the current GCF cell, the **EXTERNAL_WIRE_LOAD_MODEL** construct specifies the name of the wire load model which should be used in conjunction with the **EXTERNAL_FANOUT** construct to compute the external load capacitance.

This construct is a Level 1 construct because it requires a separate source of wire load models for proper interpretation. The wire load models are not defined in GCF, simply referenced by name.

Syntax

external_wire_load_model_spec::=
(*label?* **EXTERNAL_WIRE_LOAD_MODEL**
*library_name? wire_load_model_name port_instance** **)**

wire_load_model_name ::= QSTRING

Example

```
(parasitics
  (environment
    (level 1
      (external_wire_load_model “custom_wlms” “chip_wlm”
        out1)
    )
  )
)
```

In this example, the external wire load model “chip_wlm” from the “custom_wlms” library is assigned to output pin *out1*.

Wire Load Model

.A wire load model can also be assigned for the nets contained within particular instances or cell types, using the **WIRE_LOAD_MODEL** construct.

This construct is a Level 1 construct because it requires a separate source of wire load models for proper interpretation. The wire load models are not defined in GCF, simply referenced by name.

Syntax

wire_load_model_spec ::= **(** *label?* **WIRE_LOAD_MODEL**
library_name? wire_load_model_name
cell_instance+ **)**
||= **(** *label?* **WIRE_LOAD_MODEL**
library_name? wire_load_model_name
cell_id **)**

Example

```
(parasitics
  (environment
    (level 1
      (wire_load_model “small_wlm” a/b a/c)
      (wire_load_model “medium_wlm” (CELLTYPE “FSM2”))
    )
  )
)
```

In this example, the wire load model “small_wlm” is assigned to instances *a/b* and *a/c*, and the wire load model “med_wlm” is assigned to the master cell type “FSM2”.

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*

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*
||= *meta_data_1*

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

Internal Loading

For an interface net that is connected to a primary port on the current GCF cell, the **INTERNAL_LOAD** construct specifies a constraint on the portion of the net capacitance that is contained within the current GCF cell.

INTERNAL_LOAD specifies the capacitance allowed inside the current GCF cell, while **EXTERNAL_LOAD** specifies the capacitance that exists outside the current GCF cell.

Syntax

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

The **INTERNAL_LOAD** constraint can be specified for both input and output ports. If no *port_instance* is specified, the specification applies by default to all primary ports. The *capacitance_value* follows the conventions for *min_max* described in “Value Types” on page 48.

Loading

The load limit, or constraint on the capacitance, of a net that is entirely contained within the current GCF cell (not connected to any primary ports of the current GCF cell) can be specified in terms of an explicit capacitance

value using the **LOAD** construct. The *capacitance_value* follows the conventions for *min_max* described in “Value Types” on page 48.

Syntax

load_spec ::= **(** *label?* **LOAD** *capacitance_value* *port_instance_or_master** **)**

The constraint on the capacitance of a non-interface net can be specified on any port connected to the net. A default load limit can be specified by omitting port instances, and it applies to all nets entirely contained within the current GCF cell.

A master-based default load limit can also be specified using the *port_master* form of *port_instance_or_master*. The master-based default load limit applies to all nets which are entirely contained within the current GCF cell and are connected to an occurrence of a port corresponding to the *port_master*.

Precedence Rules

- Usually, the load limit is specified in the library. If the load limit is specified in both the library and the GCF file, the more restrictive constraint will be used.
- Explicit load limits have higher precedence than master-based default load limits, which have higher precedence than normal default load limits.
- If different constraints affect several ports connected to the same net, the most restrictive constraint will be used.

Internal Fanout

For an interface net that is connected to a primary port on the current GCF cell, the **INTERNAL_FANOUT** construct specifies a constraint on the number of ports which may be connected to the net inside of the current GCF cell.

INTERNAL_FANOUT specifies the number of ports that are allowed to be connected to the net inside the current GCF cell. **EXTERNAL_FANOUT** specifies the number of ports which are connected to the net outside the current GCF cell.

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. The *num_loads* value follows the conventions for *min_max* described in “Value Types” on page 48.

Fanout

The constraint on the capacitance of a net that is entirely contained within the current GCF cell (not connected to any primary ports of the current GCF cell) can be specified in terms of terms of the number of loads allowed using the **FANOUT** construct. The *num_loads* value follows the conventions for *min_max* described in “Value Types” on page 48.

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 nets entirely contained within the current GCF cell.

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 41, “Meta Data” on page 44, and “Include Files” on page 46.

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*

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*
||= *meta_data_1*

Example

```
(CELL (CELLTYPE "WORKLIB" "ALU")
  (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*

The *area_value* follows the conventions for *min_max* described in “Value Types” on page 48.

Example

```
(PRIMITIVE_AREA 0 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* **)**

The *area_value* follows the conventions for *min_max* described in “Value Types” on page 48.

Example

```
(TOTAL_AREA 0 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*

The *porosity_value* follows the conventions for *min_max* described in “Value Types” on page 48.

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 41, “Meta Data” on page 44, and “Include Files” on page 46.

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*

power_cnstr_spec_1 ::= **(LEVEL 1** *power_cnstr_1+* **)**

power_cnstr_1 ::= *power_cnstr_case*
||= *meta_data_1*

Example

```
(CELL ()
  (SUBSET POWER
    (AVG_CELL_POWER * 50 a/b)
  )
)
```

Power Constraints

Average Cell Power

The average power consumed by a 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*

The *power_value* follows the conventions for *min_max* described in “Value Types” on page 48.

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. The *power_value* follows the conventions for *min_max* described in “Value Types” on page 48.

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 193 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 34 for details on how the hierarchy delimiter character is established.

PATH_EXPR — is a PATH that can also contain one or more WILDCARD characters, to match arbitrary substrings between hierarchy delimiters. As with PATH, the names of levels in the design hierarchy must be separated by the hierarchy delimiter character, and a WILDCARD only matches names within that level of the design hierarchy, not across levels of the design hierarchy. A path expression 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 WILDCARD character must not be escaped or it loses its meaning as a matching character.

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 70 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*
||= *current_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* **)**
current_scale ::= **(CURRENT_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 ::= *min_number max_number*

r_min_and_max ::= *r_min_number r_max_number*

min_number ::= NUMBER
max_number ::= NUMBER
r_min_number ::= RNUMBER
r_max_number ::= RNUMBER

min_max ::= NUMBER
||= *min_value max_value*

r_min_max ::= RNUMBER
||= *r_min_value r_max_value*

```
min_value              ::= number_or_place_holder
max_value              ::= number_or_place_holder
r_min_value            ::= r_number_or_place_holder
r_max_value            ::= r_number_or_place_holder

number_or_place_holder ::= NUMBER
                       ||= *

r_number_or_place_holder ::= RNUMBER
                         ||= *

rise_fall              ::= NUMBER
                       ||= rise_value fall_value

r_rise_fall            ::= RNUMBER
                       ||= r_rise_value r_fall_value

rise_value             ::= number_or_place_holder
fall_value             ::= number_or_place_holder
r_rise_value           ::= r_number_or_place_holder
r_fall_value           ::= r_number_or_place_holder

rise_fall_min_max      ::= NUMBER
                       ||= rise_value fall_value
                       ||= rise_min_value rise_max_value
                                fall_min_value fall_max_value

r_rise_fall_min_max    ::= RNUMBER
                       ||= r_rise_value r_fall_value
                       ||= r_rise_min_value r_rise_max_value
                                r_fall_min_value r_fall_max_value

rise_min_value         ::= number_or_place_holder
rise_max_value         ::= number_or_place_holder
fall_min_value         ::= number_or_place_holder
fall_max_value         ::= number_or_place_holder
r_rise_min_value       ::= r_number_or_place_holder
r_rise_max_value       ::= r_number_or_place_holder
r_fall_min_value       ::= r_number_or_place_holder
r_fall_max_value       ::= r_number_or_place_holder
```

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*
||= *lifetime*
||= *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* **)**

lifetime ::= **(** *label?* **LIFETIME** *lifetime_value* **)**

lifetime_value ::= *min_max*

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

env_globals_1 ::= *env_globals_case*
||= *meta_data_1*

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 ::= *slew_mode*
||= *primary_waveform*
||= *extension*
||= *meta_data*

slew_mode ::= **(** *label?* **SLEW_MODE** *slew_mode_value* **)**

slew_mode_value ::= **WORST**
||= **CRITICAL**

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** *edge_position* **)**
neg_edge ::= **(NEGEDGE** *edge_position* **)**

edge_position ::= *ideal_edge*
||= *ideal_edge_with_jitter*
||= *edge_range*

ideal_edge ::= RNUMBER
||= *placeholder*

ideal_edge_with_jitter ::= *ideal_edge jitter_spec*

jitter_spec ::= **(** **JITTER** *jitter_value* **)**

jitter_value ::= NUMBER
||= *neg_jitter pos_jitter*

neg_jitter ::= NUMBER
pos_jitter ::= NUMBER

edge_range ::= r_min_and_max (archaic)

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*
||= *meta_data_1*

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

parent_waveform_name ::= QSTRING

derived_waveform_option ::= *period_multiplier*
||= *period_divisor*
||= *derived_edges*
||= *phase_shift*
||= *jitter_adjustment*
||= *invert*

period_multiplier ::= **(** **PERIOD_MULTIPLIER** *period_multiplier_value* **)**

period_divisor ::= **(** **PERIOD_DIVISOR** *period_divisor_value duty_cycle_value?* **)**

derived_edges ::= **(** **EDGES** *derived_edge_list* **)**

derived_edge_list ::= *derived_pos_pair+*
||= *derived_neg_pair+*

derived_pos_pair ::= *derived_pos_edge derived_neg_edge*

derived_neg_pair ::= *derived_neg_edge derived_pos_edge*

derived_pos_edge ::= **(POSEDGE** *derived_edge* **)**

derived_neg_edge ::= **(NEGEDGE** *derived_edge* **)**

derived_edge ::= *edge_num derived_edge_shift?*

derived_edge_shift ::= **(PHASE_SHIFT** *edge_shift_value* **IDEAL***?* **)**

phase_shift ::= **(PHASE_SHIFT** *phase_shift_value* **IDEAL***?* **)**

jitter_adjustment ::= **(JITTER_ADJUSTMENT** *edge_pair_list* **)**

invert ::= **INVERT**

period_multiplier_value ::= DNUMBER
period_divisor_value ::= DNUMBER
duty_cycle_value ::= NUMBER
edge_num ::= DNUMBER
edge_shift_value ::= RNUMBER

phase_shift_value ::= *r_rise_fall*

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 ::= *untyped_cell_instance*
||= *typed_instance_list*

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

typed_instance_list ::= **(INSTANCE** *untyped_cell_instance+* **)**

port_instance ::= *untyped_port_instance*
||= *typed_port_instance*

untyped_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 an *untyped_port_instance* */

typed_port_instance ::= *typed_port_list*
||= *typed_pin_list*

typed_port_list ::= **(PORT** *untyped_port_instance+* **)**

typed_pin_list ::= **(PIN** *untyped_port_instance+* **)**

net ::= *untyped_net*
||= *typed_net_list*

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

typed_net_list ::= **(NET** *untyped_net+* **)**

typed_waveform_list ::= **(WAVEFORM** *waveform_name+* **)**

typed_instance_expr ::= **(INSTANCE_EXPR** PATH_EXPR **)**

typed_port_expr ::= **(PORT_EXPR** PATH_EXPR **)**

typed_pin_expr ::= **(PIN_EXPR** PATH_EXPR **)**

typed_net_expr ::= **(NET_EXPR** PATH_EXPR **)**

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

port_master ::= **(** *cell_id scalar_port* **)**

port_instance_or_master ::= *port_instance*
||= *port_master*

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
                   ||= clock_arrival_spec
                   ||= arrival_spec
                   ||= required_spec
                   ||= external_delay_spec
                   ||= driver_spec
                   ||= input_slew_spec
                   ||= extension

clock_spec         ::= ( label? CLOCK waveform_name clock_root+ )

clock_root         ::= general_port_instance

clock_arrival_spec ::= ( label? CLOCK_ARRIVAL
                           clock_arrival_value
                           clock_arrival_item+ )

clock_arrival_value ::= r_rise_fall_min_max

clock_arrival_item ::= clock_root
                   ||= clock_leaf
                   ||= waveform_name
                   ||= typed_waveform_list

clock_leaf         ::= port_instance

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

arrival_waveform_edge ::= ( waveform_edge_identifier waveform_name )
```

```
arrival_value       ::= r_rise_fall_min_max
                    ||= ( waveform_edge_identifier r_min_max )          (archaic)

required_spec       ::= ( label? required_keyword
                              required_waveform_edge
                              required_value
                              port_instance* )

required_keyword    ::= REQUIRED
                    ||= DEPARTURE

required_waveform_edge ::= ( waveform_edge_identifier waveform_name )

required_value      ::= target_required_value

target_required_value ::= setup_rise_fall hold_rise_fall
                    ||= ( waveform_edge_identifier setup_value hold_value )  (archaic)

setup_rise_fall     ::= r_rise_fall
hold_rise_fall      ::= r_rise_fall

setup_value         ::= RNUMBER
hold_value          ::= RNUMBER

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

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

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?
                              port_instance* )

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_min_max*

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*
||= *meta_data_1*

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*
||= *slew_limit_spec*
||= max_transition_time_spec (archaic)
||= *extension*

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_mode_spec*
||= *clock_delay_spec*
||= *clock_uncertainty_spec*
||= *meta_data_1*

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

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

thru_spec ::= **(THRU** *port_instance* **)**

arc_spec ::= **(ARC** *port_instance port_instance* **)**

endpoints_spec ::= *from_spec*
||= *to_spec*
||= **(BETWEEN**? *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*
||= *typed_waveform_name_list*
||= *typed_port_expr*
||= *typed_pin_expr*
||= *typed_instance_expr*

from_to_thru_spec ::= **(PATHS** *from_to_thru_item+* **)**

from_to_thru_item ::= *from_opt_edge_spec*
||= *to_opt_edge_spec*
||= *thru_all_items_spec*

from_opt_edge_spec ::= *from_spec*
||= **(FROM** *from_item_edge+* **)**

to_opt_edge_spec ::= *to_spec*
||= **(TO** *to_item_edge+* **)**

from_item_edge ::= **(** *edge_identifier from_to_item+* **)**
to_item_edge ::= **(** *edge_identifier from_to_item+* **)**

thru_all_items_spec ::= **(THRU_ALL** *thru_any_item_spec+* **)**

thru_any_item_spec ::= *thru_item*
||= **(THRU_ANY** *thru_item+* **)**

thru_item ::= *port_instance*
||= *net*
||= *typed_port_expr*
||= *typed_pin_expr*
||= *typed_net_expr*
||= *port_instance_edge*

port_instance_edge ::= **(** *edge_identifier port_instance* **)**

disable_spec_0 ::= *disable_item_spec_0*
||= *disable_endpoints_spec_0*
||= *disable_from_to_thru_spec_0*

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

disable_item_0 ::= *port_instance*
||= *cell_instance*
||= *typed_port_expr*
||= *typed_pin_expr*
||= *typed_instance_expr*
||= *arc_spec*
||= *preset_clear_spec*
||= *reentrant_paths_spec*

preset_clear_spec ::= **(PRESET_CLEAR_ARCS** *true_false* **)**

reentrant_paths_spec ::= **(REENTRANT_PATHS** *true_false* **)**

true_false ::= **TRUE**
||= **FALSE**

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

disable_option ::= *timing_check*
||= *edge_identifier*

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

disable_from_to_thru_spec_0 ::= **(** *label?* **DISABLE** *from_to_thru_spec+ disable_option** **)**

```
disable_spec_1 ::= disable_cell_spec_1
                ||= disable_edges_spec_1                                (archaic)

disable_cell_spec_1 ::= ( label? DISABLE disable_cell_path_spec+ )

disable_cell_path_spec ::= disable_instance_spec
                       ||= disable_master_spec

disable_instance_spec ::= ( INSTANCE untyped_cell_instance+ )

disable_master_spec ::= ( MASTER cell_id )

multi_cycle_spec_0 ::= multi_cycle_endpoints_spec_ 0
                   ||= multi_cycle_from_to_thru_spec_0

multi_cycle_spec_0 ::= ( label? MULTI_CYCLE multi_cycle_endpoints_param_list )

multi_cycle_endpoints_param_list ::= endpoints_spec+ multi_cycle_option+
                                 ||= 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_from_to_thru_spec_0 ::= ( label? MULTI_CYCLE
                                        multi_cycle_from_to_thru_param_list )

multi_cycle_from_to_thru_param_list ::= from_to_thru_spec+ multi_cycle_option+
                                    ||= multi_cycle_option+ from_to_thru_spec+

path_delay_spec_0 ::= path_delay_endpoints_spec_ 0
                  ||= path_delay_from_to_thru_spec_0

path_delay_endpoints_spec_0 ::= ( label? PATH_DELAY
                                    path_delay_value
                                    endpoints_spec+ )

path_delay_value ::= rise_fall_min_max
                 ||= path_delay_single_value                            (archaic)

path_delay_from_to_thru_spec_0 ::= ( label? PATH_DELAY path_delay_value from_to_thru_spec+ )

slew_limit_spec ::= ( label? SLEW_LIMIT slew_value port_instance_or_master* )
```

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

borrow_value ::= NUMBER

borrow_item ::= *port_instance*
||= *cell_instance*
||= *waveform_name*

clock_mode_spec ::= **(** *label?* **CLOCK_MODE** *clock_mode_value* **)**

clock_mode_value ::= **IDEAL**
||= **ACTUAL**

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

clock_delay_root ::= *untyped_port_instance*
||= **(** *cell_instance input_port output_port* **)**
||= *waveform_name*

leaf_spec ::= *clock_mode_value*
||= *default_leaf_spec*
||= *explicit_leaf_spec*

default_leaf_spec ::= **(** *default_leaf_option+* **)**

default_leaf_option ::= *insertion_delay_spec*
||= *clock_skew_spec*
||= *clock_slew_spec*

explicit_leaf_spec ::= **(** *explicit_leaf_option* clock_delay_leaf+* **)**

explicit_leaf_option ::= *insertion_delay_spec*
||= *internal_insertion_delay_spec*
||= *clock_slew_spec*

clock_delay_leaf ::= *clock_leaf*
||= *data_leaf*

data_leaf ::= **(** **DATA** *port_instance+* **)**

insertion_delay_spec ::= **(** **INSERTION_DELAY**
insertion_delay_value **)**

internal_insertion_delay_spec ::= **(** **INTERNAL_INSERTION_DELAY**
insertion_delay_value **)**

clock_skew_spec ::= **(** **SKEW** *skew_value* **)**
clock_slew_spec ::= **(** **SLEW** *slew_value* **)**

insertion_delay_value ::= *rise_fall_min_max*
skew_value ::= *rise_fall_min_max*
slew_value ::= *rise_fall_min_max*

clock_uncertainty_spec ::= **(** *label?* **CLOCK_UNCERTAINTY**
*clock_uc_option**
clock_uc_value
clock_uc_item **)**

clock_uc_option ::= *clock_uc_calc_option*
||= *clock_uc_mode_option*

clock_uc_calc_option ::= **ABSOLUTE**
||= **INCREMENT**

clock_uc_mode_option ::= **IDEAL**
||= **ACTUAL**

clock_uc_value ::= *r_min_max*

clock_uc_item ::= *target_clock_uc_item+*
||= *target_clock_uc_item_edge*
||= *inter_clock_uc_item*

target_clock_uc_item ::= *waveform_name*
||= *typed_waveform_list*
||= *clock_root*
||= *clock_leaf*
||= *clock_leaf_instance*

clock_leaf_instance ::= *cell_instance*

target_clock_uc_item_edge ::= **(** *waveform_edge target_clock_uc_item+* **)**

inter_clock_uc_item ::= **(** **BETWEEN** *inter_clock_from inter_clock_to* **)**

inter_clock_from ::= **(** **FROM** *inter_clock_from_to_item* **)**
inter_clock_to ::= **(** **TO** *inter_clock_from_to_item* **)**

inter_clock_from_to_item ::= *waveform_name*
||= *waveform_edge*

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

waveform_edge_identifier ::= **POSEDGE**
||= **NEGEDGE**

edge_identifier ::= **POSEDGE**
||= **NEGEDGE**
||= **ANYEDGE**
||= **0z**
||= **z1**
||= **1z**
||= **z0**

Archaic Timing Exceptions

The archaic timing exceptions are defined as follows:

thru_edge_spec ::= (**THRU** port_instance_edge) (archaic)

arc_edges_spec ::= (**ARC** port_instance_edge port_instance_edge) (archaic)

thru_all_spec ::= (**THRU_ALL** port_instance port_instance+) (archaic)

thru_all_edges_spec ::= (**THRU_ALL** port_instance_edge
port_instance_edge+) (archaic)

disable_edges_spec_1 ::= (label? **DISABLE** disable_edges_path_spec+
timing_check?) (archaic)

disable_edges_path_spec ::= thru_edge_spec
||= arc_edges_spec
||= thru_all_edges_spec (archaic)

multi_cycle_spec_1 ::= multi_cycle_thru_spec_1 (archaic)

multi_cycle_thru_spec_1 ::= (label? **MULTI_CYCLE**
multi_cycle_option+
multi_cycle_thru_path_spec_1+) (archaic)

multi_cycle_thru_path_spec_1 ::= arc_spec
||= thru_spec
||= thru_all_spec (archaic)

path_delay_single_value ::= (timing_check waveform_edge_identifier NUMBER)

path_delay_spec_1 ::= path_delay_path_spec_1 (archaic)

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

path_delay_path_spec_1 ::= arc_spec
||= thru_spec
||= thru_all_spec (archaic)

max_transition_time_spec ::= (label? **MAX_TRANSITION_TIME**
rise_fall port_instance*) (archaic)

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*

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

capacitance_value ::= *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_wire_load_model_spec*
::= *wire_load_model_spec*
||= *meta_data_1*

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

num_loads ::= *min_max*

external_wire_load_model_spec ::= **(** *label?* **EXTERNAL_WIRE_LOAD_MODEL**
library_name? wire_load_model_name
*port_instance** **)**

wire_load_model_name ::= QSTRING

wire_load_model_spec ::= **(** *label?* **WIRE_LOAD_MODEL**
library_name? wire_load_model_name
cell_instance+ **)**

```
                    ||= ( label? WIRE_LOAD_MODEL
                              library_name? wire_load_model_name
                              cell_id )

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

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

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

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
                           ||= meta_data_1

internal_fanout_spec ::= ( label? INTERNAL_FANOUT num_loads port_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*

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*
||= *meta_data_1*

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

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
                  ||= meta_data_1

power_cnstr_case  ::= ( CASE IDENTIFIER power_cnstr_case_spec+ )

power_cnstr_case_spec ::= power_cnstr_spec_0
```

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Appendix 1

Cadence-Specific Extensions

TLF Files

The locations of the Timing Library Format (TLF) files that 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*
||= *tlf_files_extension*
||= *extension*
||= *meta_data*

tlf_files_extension ::= **(EXTENSION** “TLF_FILES” **(** *file_name+* **))**
||= **(EXTENSION** “CTLF_FILES” **(** *file_name+* **))**

file_name := IDENTIFIER

The TLF_FILES extension name is preferred; the CTLF_FILES extension name is supported for backward compatibility. For either extension name, the list of files can refer to files containing clear text, compiled, or encrypted forms of TLF.

The file names can be relative or absolute path names. For GCF 1.3 and higher, relative path names are interpreted with respect to the GCF file which contains the extension, not with respect to the directory in which the program which is reading the GCF is invoked.

Previous versions of GCF were with respect to the directory in which the program which was reading the GCF was invoked.

Example

```
(GLOBALS_SUBSET ENVIRONMENT
  (EXTENSION “CTLF_FILES”
    (lib/mylib.ctlf
     lib/ram1.ctlf
     lib/ram2.ctlf
     ../lib2/ram3.ctlf
    )
  )
)
```