# General Constraint Language (Working Draft)

## Section 1

## Overview of this standard

### 1.1 Objectives of this document

The intent of this document is to serve as a complete specification of the General Constraint Language (GCL). This document contains:

— The formal syntax and semantics of all GCL constructs.
— Non-normative usage examples.

### 1.2 Synopsis of contents

A synopsis of the sections of this manual is presented here for a quick reference.

- **Section 2. Introduction**
  This section discusses the objectives of GCL.
- **Section 3. Conventions and Basic Features**
  This section dicusses conventions used for describing commands and basic features of the command language.
- **Sections 4-9. Command Descriptions**
  These sections contain the descriptions of individual commands, organized according to the same general categories used in the GCF specification.
- **Section 10. Discussion of Issues**
  This section contains informal discussion of remaining open issues.

### 1.3 Examples are informative

Examples are shown throughout this document. These examples are ***informative***—they are intended to illustrate the usage in a simple context, and do not define the full syntax.

# Section 2

# Introduction

## 2.1 Background

Today, designers who wish to describe constraints on the performance of their design, and the environment in which the design is intended to operate, are faced with a multitude of possible formats for the description. Most of these formats are proprietary and can only be used by a limited number of tools.

Designers typically must enter several different versions of the constraints, where each version has subtle differences in the supported semantics. These versions must be maintained as requirements for the design change. The constraint descriptions often represent a significant investment in time and resources. These factors make it relatively difficult to achieve inter-operability between all of the different types of tools which are required in the course of implementing and verifying a design. They also reduce competition between tool vendors, because performing benchmarks to compare different tools, or actually switching from one tool to another is a major task.

Reuse of the intellectual property (IP) in designs is becoming an increasingly important goal, with many companies now offering IP components for sale. For these companies, lack of a standard constraint format is a serious barrier to entry, because they must support many different constraint formats in order for their components to be used successfully in the wide variety of customer environments.

## 2.2 Purpose

The General Constraint Language (GCL), is an open standard co-developed by Open Verilog International (OVI) and VHDL International (VI). GCL is intended to address the problem of initial constraint entry at the point in the design process where a register-transfer level (RTL) description of a portion of the design is available.

GCL is primarily focused on the types of constraints and boundary conditions which are required for RTL->gate level synthesis. Many of these constraints are also useful for other types of tools, such as estimators, cycle simulators, floorplanners, placers, and routers.

GCL supports specifying constraints on both the design as a whole, and on various portions of the design. Not all parts of the design will necessarily be described using synthesizable RTL. For example, a hard macro implementing a memory usually is not synthesized. However, it may be useful to specify constraints for non-synthesized parts of the design, in order to control module generators or automatic selection from several different implementations of a complex macro.

## 2.3 GCL vs GCF

GCL is a constraint entry language which is intended to match the RTL description of the design. It includes a number of macro operators and pattern-matching capabilities which make it easier for a designer to describe the intended the performance of the design. As a command language, the format is also relatively free-form: unrelated constraint commands may appear in any order and consecutive commands may refer to very different parts of the design. These features make it relatively complex for a tool to support GCL.

The General Constraint Format (GCF) is a constraint interchange language which is intended to match the gate-level description of the design. It is a much more restricted format designed to make it relatively easy for a tool to support.

The distinction between the RTL description and the gate-level description of a design is an important one. Often a designer will modify the hierarchy of the RTL description during the course of synthesis, which introduces changes in the names which must be used to refer to particular elements in the design. The synthesis tool, because it is responsible for modifying the hierarchy, will usually take care of preserving the constraints on design elements as those elements are moved from one portion of the hierarchy to another. However, it would be relatively difficult for other tools to relate the pre-synthesis constraints to the gate-level netlist.

We expect GCL to be read by synthesizers, estimators, high level floorplanners, module generators, and cycle simulators. It may also be used by other tools such as timing analyzers, in order to handle mixed-level designs where some portions are RTL which has not yet been synthesized, and other portions are at the gate-level.

Synthesis tools are expected to bridge the gap between the RTL description and the gate-level description by writing the constraints in GCF form.

We expect GCF to be read by gate-level timing analyzers, detailed floorplanners, placers, routers, delay calculators, and incremental logic optimization tools.

## 2.4 GCL versus Extension Languages

GCL is not intended to be used as an extension language. It is a simple command language, with a precisely defined syntax and semantics for a finite number of commands and their options.

Extension languages are typically used in tools to provide users with a sophisticated programming environment, including variables, flow of control constructs (such as if-then-else, for, while), functions, operating system interfaces, and direct interfaces to various tool capabilities.

This kind of programming environment, while very powerful, tends to be very tool-specific. There have been attempts in the past to develop a standard extension language (Scheme), which met with limited acceptance.

The relationship between GCL and the particular extension language used in a given tool is not defined by this standard. However, we expect that a common use will be for a designer to enter GCL commands into one or more standalone files. Then, in the extension language script used to control the flow of operations in the tool, the user would issue a tool-specific command to read the GCL file and apply the constraints described in that file to a particular part of the design.

This approach intentionally separates the commands used to describe constraints (design-specific information) from the extension language and commands used to control the tool (tool-specific information). The purpose of this separation is to provide portability for the constraints across multiple tools, and to facilitate adoption of GCL. Instead of requiring tool vendors to implement a whole new (and probably incompatible) programming environment, we just require that they be able to read a command file. By providing a reference parser for that command file, we can make it even easier for a vendor to implement support for GCL.

## 2.5 Overview of GCL Contents

GCL contains constraints on the timing, area, power, and parasitics of the design. It also contains specifications of the operating environment and boundary conditions which are to be assumed in doing analysis or optimization. Finally, GCL contains a number of synthesis-specific directives, which are used to provide more information about the hardware implementation which the designer intends to be inferred from the RTL description. These directives are generic in the sense that they are not tied to any particular tool’s capabilities; they describe certain features of the desired implementation which cannot be expressed directly in the VHDL and Verilog languages today.

# Section 3

# Conventions and Basic Features

## 3.1 Conventions used in this document

The remainder of this document is organized into sections, each of which focuses on some specific area of the language. There are subsections within each section to discuss with individual constructs and concepts. The discussion begins with an introduction and an optional rationale for the construct or the concept, followed by syntax and semantic description, followed by some examples and notes.

## 3.2 Syntactic description

The following conventions are used for specifying command syntax:

a) Boldface words are used to denote reserved keywords and punctuation marks as required parts of the syntax. Reserved keywords include command names, option names, and macro names. For example, in the following, the words **gcl_waveform** and **-period** are reserved keywords.

**gcl_waveform** wave_name **-period** period

(Note: The example syntax used here has been simplified for illustrative purposes.)

a) Lowercase words are used to denote command arguments for which a specific value is to be provided. For example, in the above example, “wave_name” is an argument for which the name of a waveform is to be substituted, and “period” is an argument for which a numeric value is to be substituted. An example of a command formed according to this syntax is:

```
gcl_waveform wav02 -period 20
```

b) Square brackets enclose optional items. The square brackets are not part of the actual command. For example:

**gcl_unset_broken_arc** [ **-all** ]

Example commands formed according to this syntax are:

```
gcl_unset_broken_arc
gcl_unset_broken_arc -all
```

c) A vertical bar separates alternative items. The list of alternative items is enclosed in braces if exactly one item must be used, and in square brackets if at most one item must be used. The braces or brackets are not part of the actual command. For example:

**gcl_set_external_delay** value nets [ **-max** | **-min** ] { **-posedge** wave | **-negedge** wave }

The following are examples of commands formed according to this syntax:

```
gcl_set_external_delay 3.4 Ain -posedge wav02
gcl_set_external_delay 0 resetWB -max -negedge zwave
gcl_set_external_delay -5 B[*] -min -posedge zclk
```

The main text uses *italicized* font when a term is being defined, and `constant-width` font for examples and file names.

## 3.3 Command structure

Commands have the following syntactical parts:

- command name
- command arguments : any argument that is not introduced by an option keyword.
- options. An option consists of an “option keyword” which may or may not be followed by an “option argument”

For example:

```
gcl_set_external_delay 3.0 int0 -max -posedge ck1
                       |   |    |     |      |___option argument
                       |   |    |     |__________option keyword
                       |   |    |________________option keyword
                       |   |_____________________command argument
                       |_________________________command argument
```

Command names consist of a single word, which may contain underscores.

Option keywords begin with a minus sign followed by an alphabetic character (thereby distinguishing them from numeric arguments).

Command arguments have a fixed order in the command relative to each other. Thus, above, the arrival time “3.0” must precede the port name “int1”.

Options may appear in arbitrary order relative to each other and to command arguments. Thus, the following commands are equivalent to the above example command:

```
gcl_set_external_delay 3.0 -posedge ck1 -max int0
gcl_set_external_delay -max 3.0 -posedge ck1 int0
```

Any given option keyword either has a mandatory option argument or does not take an argument. This makes it possible to parse command arguments even if they come after an option.

Any given option keyword must be used at most once in a command.

## 3.4 Lists as arguments

Some command arguments and option arguments can be lists of items. When an argument is a list consisting of more than one item, the items are separated by spaces, and the list of items is enclosed in braces. Braces used to delimit a list do not appear in the syntax specification, but are mandatory in the actual command if the list consists of more than one item. Braces are optional if the list consists of one item. For example:

```
gcl_set_external_delay 3.0 { int0 int1 } -max -posedge ck1
```

This is identical to the previous example, except that the one-item list “`int0`” has been replaced by a list of two items, “`{ int0 int1 }`”.

An item in a list may itself be a list (for example, see the **gcl_waveform** command). In the event that it is necessary to represent a list whose first and only element is a list, two levels of braces must be used; e.g. `{{0 1}}`.

### 3.4.1 Strings as arguments

In a few commands, a string argument is permitted to contain blank space. A string argument that contains blank space must be delimited by double-quote marks. For example:

```
gcl_version "Cadence Version 0.6"
```

Double-quote marks are optional for a string that does not contain blank space. Thus, the following two commands are equivalent:

```
gcl_design "ipc_counter"
gcl_design ipc_counter
```

## 3.5 Command and Option Name Conventions

All command names begin with the letters "`gcl_`". This convention has been adopted to eliminate any possible confusion of GCL commands with command names in a tool's extension language.

The use of underscores in forming command names and option names has been guided by the following rules:

a) An underscore is used after the initial `gcl` of a command name.

b) In all other places, an underscore is used if and only if, in an English sentence, the two parts being separated would normally be written as two words separated by a space. If a word is normally hyphenated in English, an underscore is *not* used in place of the hyphen.

Examples:

- The option name `-posedge` does not include an underscore because "pos" is not an English word and so "pos edge" would not be written as two separate words in an English sentence.
- The option name `-setup` does not include an underscore because "setup" is normally written as a single word in English. (Even if you think it should be spelled "set-up" with a hyphen, the underscore would still not be called for by the above convention.)

The use of plurals in command and option names has been avoided, even when multiple objects are referenced. For example, in the **gcl_exception_path** command, the option which specifies a list of one or more instances is named **-from_instance** and not **-from_instances**.

### 3.5.1 Command Names with "set" and "unset" and "remove"

Command names which have the effect of setting an attribute of some object generally include the word"set" in their names. For these commands, there is usually a matching "unset" command for removing the setting. For example, the **gcl_set_external_delay** command sets the external dealay attribute of a port instance, and the **gcl_unset_external_delay** command removes this attribute.

Commands which are seen as creating a new object or new data structure do not include any special word. (The word "create" could have been used in these commands but was rejected as too cumbersome.) For such commands, there is often a corresponding command with the word "remove" which removes or deletes the created object or data structure. For example, the **gcl_exception_path** command creates a point-to-point timing exception, and the **gcl_remove_exception_path** command removes a previously created exception.

## 3.6 Identifiers and Case Sensitivity

All GCL keywords (including command names, option names, reserved option values and macro names) are case-insensitve.

All other identifiers (such as the names of user-defined waveforms and constraints and the names of cells, instances, nets and ports) are case-sensitive and hence must be specified with the identical case throughout a GCL command session, regardless of whether or not the underlying tool treats them as case sensitive.

Legal characters for identifier names (excepting GCL keywords) are:

- upper and lower case alphabetic and numeric characters: A-Z, a-z, 0-9
- the hierarchy delimiter character
- the bus delimiter character when used in a syntactically correct bus and bus range specification
- any non-white-space character if preceded by the escape character (backslash)

Example:

If the hierarchy delimiter is `/` and the bus delimiter characters are `[` and `]`, the following are legal identifiers:

```
ABC
u235/AbXy
u235/SYNOUT[3]
01/03/05/68a[5:1]
smp\_over\_QN
\$3135
```

and the following are illegal:

```
u235.AbXy
u235/SYNOUT(3)
01/03/05/68a[5:3:1]
smp_over_QN
$3135
abc[x]
[55]abc
```

The hierarchy delimiter and the bus delimiter characters should be used only for specifying hierarchy and buses. Any other use, even if escaped, is an error.

## 3.7 Netlist references: wildcards and macros

Many GCL commands have arguments that are netlist objects, such as instances, nets, or port instances. At the RTL level of design the exact names of internal instance pins may not be known. Therefore, wildcards and macros are provided for referring to netlist objects.

### 3.7.1 Wildcards

The only wildcard character is *. This character will match any consecutive sequence of zero or more non-white-space characters, except the hierarchy separator. The exclusion of the hierarchy separator means that this wildcard

cannot be used to match arbitrary hierarchical names. For example, the hierarchical instance name u1/u2 will be matched by the wildcard expressions "`u1*/u*`" and "`*/*`", but not by "`*`" or "`*u2`".

of nonblank characters, including the empty sequence.

The range of selection of the wildcard is determined by the permitted argument types of the command. For example, if the arguments of a command are specified as external input ports of the design, then a wildcard in this position will match only external input ports.

A bus name cannot be used in place of a list of signal names. If a command expects a list of net names as an argument, the argument must be given in a form such as `A[*]` or `A(*)`, where `A` is the name of a bus.

A bus range may be indicated in format such as "`A[3:0]`". This is equivalent to specifying the list "`A[3] A[2] A[1] A[0]`". The left and right bounds of the range must be specified as integers and must not be omitted.

Note that only numeric values are permitted as bus indices. GCL does not provide for defining variables that can be used in place of a numeric value.

### 3.7.2 Macros

The following macros are available:

**all_inputs()** all external input ports of the design

**all_outputs()** all external output ports of the design

**all_inputs(**clk**)** all external input ports in the transitive fanin of data input pins of sequential instances clocked by the clock clk

**all_outputs(**clk**)** all external output ports in the transitive fanout of data output pins of sequential instances clocked by the clock clk

**all_se()** all sequential instances

**all_ff()** all edge-triggered flip-flop instances

**all_latch()** all level-sensitive latch instances

**all_se(**clk**)** all sequential instances clocked by the clock clk

**all_ff(**clk) all flip-flop instances clocked by the clock clk

**all_latch(**clk**)** all latch instances clocked by the clock clk

**fanout_instances(**x**)** all instances in transitive combinational fanout of net x

**fanout_ports(**x**)** all port instances in transitive combinational fanout of net x

**fanin_instances(**x**)** all instances in transitive combinational fanin of net x

**fanin_ports(**x**)** all port instances in transitive combinational fanin of net x

Note: The above four macros recognize the same path arcs that are seen as combinational arcs when timing of the design is analyzed. Thus the output of this macro is affected by commands such as **gcl_set_broken_arc** that disable or enable path arcs.

**data_input()** data output pin(s) of sequential instance

**clock()** clock pin(s) of a sequential instance

**data_output()** data output pin(s) of a sequential instance

**set_reset()** set and reset pin(s) of a sequential instance

These macros can in some cases be combined. For example, `all_se(clk).data_input()` refers to the data input pins of all sequential instances clocked by the clock `clk`.

```
/*
```

comment by tom schaefer:
We have a problem if parentheses are chosen as the index characters for bus names (see the DELIMITERS construct of GCF), as then there could be ambiguity between macros and net names.
It seems we either need to have a truly distinctive syntax for macros, or else provide a means for escaping identifiers that conflict with macros. The first alternative is clearly better, because simply escaping conflicting identifiers can result in incompatibility when new macros are defined. One idea: require all macros to begin with a + sign.

Question: In the above, can clk be a clock insertion port, or a waveform, or either?

```
*/
```

# Section 4

# Header Commands

## 4.1 Header Commands

The commands in this section correspond to items in the GCF header section. The header commands are not nornally used in interactive command entry, but may be used in command files to permit portability between tools.

### 4.1.1 gcl_version

Syntax: **gcl_version** version_string

Declares the General Constraint Language version number. This should be the first line (other than comments) of any saved command file.

Arguments:

version_string version name

Example:

```
gcl_version "Cadence Version 0.6"
```

### 4.1.2 gcl_design

Syntax: **gcl_design** design_name

Specifies the name of the design. For documentation only. If used, it should appear at the beginning of the file, after the **gcl_version** command.

Arguments:

design_name design name

Example:

```
gcl_design ipc_counter
```

### 4.1.3 gcl_delimiter

Syntax: **gcl_delimiter** [ **-hierarchy** dhier ] [ **-bus** dbus ]

Defines hierarchy and bus index delimiters.

Arguments:

| | |
|---|---|
| dhier | Hierarchy delimiter character ("/" or "."). |
| dbus | Left and right bus index delimiter characters ("[]" or "()" or "<>"). |

Examples:

```
gcl_delimiter -hierarchy "/" -bus "()"
```

### 4.1.4 Dimensional unit commands

Syntax:

**gcl_area_scale** area_scale
**gcl_capacitance_scale** cap_scale
**gcl_length_scale** length_scale
**gcl_power_scale** power_scale
**gcl_resistance_scale** res_scale
**gcl_time_scale** time_scale
**gcl_voltage_scale** voltage_scale

Specifies the units in which area, capacitance, length, power, resistance, time and voltage are expressed, relative to the default units of square meters, farads, meters, watts, ohms, seconds and volts. The default multiplier value is 1.

Examples:

```
gcl_time_scale 1.0E-9
gcl_capacitance_scale 1e-12
gcl_power_scale .001
gcl_length_scale 1e-6
gcl_area_scale 1e-12
```

The above commands indicate that time is expressed in nanoseconds, capacitance in picofarads, power in milliwatts, length in micrometers, and area in square micrometers. If the **gcl_resistance_scale** and **gcl_voltage_scale** commands are not present, resistance and voltage will be in units of ohms and volts respectively.

## 4.2 GCF Header Constructs Not Represented in GCL

There are no GCL commands corresponding to the DATE and PROGRAM constructs of GCF. If information on date or program are desired in a GCL command file, it may be included as comments.

# Section 5

# Global Commands

## 5.1 Global Environment Commands

A command language that specifies precise limits on delay or power in a design would not be complete if it did not indicate the assumptions which are to be used in computing the delay or power. For example, to require that a path has a delay of 3 ns at 25 degrees Celsius is not the same as requiring a 3-ns delay at 85 degrees.

*Global environment parameters* are parameters such as supply voltage, temperature, and process, which can affect the calculation of delays and power consumption. Global environment parameters do not include parameters, such as load and drive, that are associated with specific instance pins.

The list of global environment parameters will depend on the delay calculation model and the power calculation model that are being used. Different delay calculation models may recognize different sets of global environment parameters. Different portions of the design may use different delay calculation models or have different settings of the global environment parameters.

A detailed treatment of delay calculation is beyond the scope of this specification. It is, therefore, our intent that the commands of this section should serve merely as a hand-off mechanism, allowing whatever parameters the delay and power calculation models require to be passed through from the user in a transparent fashion.

While some environment parameters (such as temperature and supply voltage) are easily conceptualized as a scalar quantity, others (notably process) are inherently multidimensional. For example, a user may have six different transistor models and four different metal models and may wish to specify any combination of these as the process to be targeted by the delay calculator. It is easy to imagine a timing model that requires not one, but two or more “process” parameters to capture these independent aspects of process variation. It is important that our command set should allow this flexibility to the user.

GCL offers two ways for the global environment commands to be specified. They may be specified directly using the **gcl_set_environment** command. Or they may be specified indirectly, by defining one or more “operating conditions” and then using the **gcl_set_operating_condition** command to select one of the defined operating conditions.

### 5.1.1 gcl_set_environment

Syntax: **gcl_set_environment** param_name value [ **-instance** insts | **-cell** cells ]

Specifies the value of a global environment variable. The value may be specified to apply to only certain instances or to the whole design.

Arguments:

| | |
|---|---|
| param_name | name of global environment parameter<br>GCL does not dictate a fixed list of names, but some enviroment parameters are so universal that we specify a fixed name for them. If the parameter being described is temperature, the param_name should be **temperature**, and the value should be in degrees Celsius. If the parameter being described is supply voltage, the param_name should be **voltage**, and the value should be in volts. If the parameter being represented is a scalar multiplier representing the process, the param_name should be **process**. |

| | |
|---|---|
| value | value of parameter. This value may be a list, and may consist of either numeric or alphanumeric information. |

Options:

| | |
|---|---|
| **-instance** insts | List of instances to which this parameter setting applies. |
| **-cell** cells | List of cells. The parameter setting applies to all instances of these cells. |

If neither **-instance** nor **-cell** is specified, the parameter setting applies to all instances.

Example:

```
gcl_set_environment voltage 3.3
```

most likely indicates that the supply voltage to be assumed for delay and power calculations is 3.3 volts. The exact interpretation, however, is determined by the delay and/or power models used in the design.

```
gcl_set_environment process { PSLOW NSLOW M2FAST }
```

This command indicates that the value of the "process" parameter is a list `{ PSLOW NSLOW M2FAST }`. The meaning of this list is determined by the delay and/or power models. For example, this parameter value might indicate that delays are to be modeled according to the "slow" process models for N- and P-channel transistors and according to the "fast" process model for metal layer 2 interconnect.

### 5.1.2 gcl_define_operating_condition

Syntax: **gcl_define_operating_condition** opname param_name1 value1 [ param_name2 value2 ... ]

Defines an operating condition as a collection of settings of global environment variables. The argument list may contain any number of (param_name,value) pairs, but each parameter name must be used at most once in the command.

Arguments:

| | |
|---|---|
| opname | Name of operating condition. |
| param_name1 | name of first global environment parameter |
| value1 | value of first parameter |

Any number of additional param_name, value pairs may follow. The parameter names and values are the same as those permitted by the **gcl_set_environment** command. The order in which the parameters are listed is not significant.

Example:

```
gcl_define_operating_condition worst process 1.0 voltage 3.3 temperature 85
```

Defines an operating condition named "`worst`" consisting of the settings process = 1.0, voltage = 3.3, and temperature = 85.

### 5.1.3 gcl_set_operating_condition

Syntax: **gcl_set_operating_condition** op_cond [ **-instance** insts | **-cell** cells ]

Specifies the current operating condition. The effect of this command is equivalent to using the **gcl_set_environment** command to individually set each of the parameter values comprising the definition of the operating condition.

Arguments:

op_cond — name of an operating condition previously defined with the **gcl_define_operating_condtion** command.

Options:

**-instance** insts — List of instances to which this operating condition applies.

**-cell** cells — List of cells. The operating condition applies to all instances of these cells.

If neither **-instance** nor **-cell** is specified, the operating condition applies to all instances.

Example:

```
gcl_set_operating_condition worst
```

Specifies that the previously defined `worst` operating condition applies to all instances.

### 5.1.4 gcl_voltage_threshold

Syntax: **gcl_voltage_threshold** thresh_rise thresh_fall

Specifies the voltage threshold, or trip-point, for rising and falling signals, expressed as a percentage of the low-to-high voltage swing.

Arguments:

thresh_rise — Trip-point percentage for rising signals.

thresh_fall — Trip-point percentage for falling signals.

Example:

Suppose that the ground voltage is 0 volts and the supply voltage is 3.3 volts. Thus, the total low-to-high voltage swing is 3.3 volts.

```
gcl_voltage_threshold 65 40
```

This command means that the trip-point for rising signals is .65*3.3, or 2.145 volts, and the trip-point for falling signals is .40*3.3, or 1.32 volts.

## 5.2 Timing Globals Commands

### 5.2.1 gcl_waveform

Syntax: **gcl_waveform** wavename [ **-period** period ] [ **-rise_first** edgelist | **-fall_first** edgelist ]

Defines a primary waveform, or modifies the attributes of an existing primary waveform.

If there is currently no waveform having the specified name, a new waveform is defined.

If there currently is a waveform having the specified name: (1) If period is specified, the period of the waveform is set to the new value. (2) If edgelist is specified, the previous edgelist is removed and replaced by the new one. (3) All design objects (clock insertion ports, constraints, arrival times, etc) that were previously associated with the waveform remained associated with it.

Arguments:

| | |
|---|---|
| wavename | name of waveform. If this is the same as the name of an external port of the design, that port is automatically associated with the waveform as a clock insertion port, just as if a **gcl_clock** command had been given for it. |

Options:

| | |
|---|---|
| **-period** period | period of the waveform in time units |
| **-rise_first** edgelist | list of all clock edges in the period starting at or after time 0, starting with a rising edge. A range of arrival times is indicated by a pair of numbers in braces. The earliest arrival time specified for each edge in the list must be greater than or equal to zero, and less than the period. |
| **-fall_first** edgelist | list of all clock edges in the period starting at or after time 0, starting with a falling edge |

When a range of arrival times is specified for a clock edge, this indicates uncertainty in the arrival time of the edge with respect to earlier and later edges of the same clock. It is not intended to indicate variation between the times when the same clock edge arrives at different clock insertion ports. Timing analysis should assume that any given edge of any given waveform arrives simultaneously at all clock insertion ports associated with the waveform. If it is desired to indicate variation in arrival time between different clock insertion ports, a different waveform should be associated with each clock insertion port.

Example:

```
gcl_waveform CK1 -period 20 -rise_first { 0.1 {9.8 10.2} }
```

Defines a waveform named `CK1`, having period 20. The rising edge is at time 0.1. The earliest and latest arrival times for the falling edge are 9.8 and 10.2. If the netlist has a net named `CK1`, that net is set to be a clock node having the waveform `CK1` as its associated waveform.

```
/*
```
comment by tom schaefer:
The GCF 1.0 spec defines wavename as a quoted string, which allows it to contain embedded spaces. Do we really need this luxury? How many tools can support names that contain spaces?
```
*/
```

### 5.2.2 gcl_derived_waveform

Syntax: **gcl_derived_waveform** name [ **-parent** parent ] [ **-multiplier** mult ] [ **-shift** shift ] [ **-edge_adjustment** deltas ]

Defines a derived waveform, or modifies the attributes of an existing derived waveform.

If name is not the name of a previously defined waveform, this command defines a new derived waveform. In this case, the **-parent** option must specify the name of a previously defined waveform.

If name is the name of a previously defined derived waveform, the attributes of that waveform corresponding to the options present in the command are changed to have the specified values. In this case, the **-parent** option must not be present.

It is an error for name to be the name of a previously defined primary waveform.

Arguments:

| | |
|---|---|
| name | name of derived waveform. If this is the same as the name of a net in the design, that net is automatically associated with the waveform as a clock insertion port, just as if a **gcl_clock** command had been given for it. |

Options:

**-parent** parent name of parent waveform. Wildcard not permitted.

**-multiplier** mult period multiplier

**-shift** shift phase shift (applied after period multiplier)

**-edge_adjustment** deltas edge adjustments. A list containing one item for each edge of the parent waveform. Each item is a single real number or a list of two real numbers. If a single number, it is subtracted from the left edge of the uncertainty region and added to the right edge. If two numbers, the first is subtracted from the left edge of the uncertainty region, and the second is added to the right edge. Applied after period multiplier and phase shift.

Example:

```
gcl_derived_waveform CK1A -parent CK1 -multiplier 2 -edge_adjustment
{ {0 .1} {0 .1}}
```

Defines a waveform `CK1A` derived from the parent waveform `CK1`. The time of each edge of each uncertainty region of `CK1A` is found by multiplying the time of the corresponding edge of `CK1` by the multiplier 2, and then adding the additional uncertainty .1 to the right edge of each uncertainty region.

# Section 6

# Timing Commands

## 6.1 GENERAL DISCUSSION OF TIMING CONSTRAINTS

/*

comment by tom schaefer:
I wrote this section to clarify my ideas about constraints. Much of this discussion is applicable to both GCF and GCL. Some of it may not really be relevant to the final spec.

*/

### 6.1.1 Basic Concepts

In specifying commands to describe timing constraints, it is important to have a conceptual model of what we mean by a timing constraint.

At the most basic level, a “timing constraint” is simply a requirement that a signal should travel from some specified “source” point to some specified “destination” point in a certain amount of time. Two kinds of timing constraint can be distinguished:

- max-delay constraint : requires that the signal take AT MOST a specified amount of time to get from one point to another
- min-delay constraint : requires that the signal take AT LEAST a specified amount of time to get from one point to another

Of course, it is not practical to specify timing constraints by specifying a separate constraint for each source point and each destination point. The normal practice is to associate with a timing constraint a LIST of source points and a LIST of destination points.

Also, the amount of time to travel between the source and destination points may not be a unique number. For example, one may want to specify different time requirements according to whether the signal is rising or falling at the source, and whether it is rising or falling at the destination.

Based on the foregoing considerations, we here define the notion of a “base constraint”. The base constraint is the fundamental building block in terms of which more complex kinds of constraints may be constructed. We define a “base constraint” to consist of the following:

- a list of source points
- a list of destination points
- rising and/or falling arrival times associated with each source point (at least one of rising and falling must be specified; if one of them is not specified, that signal direction is not constrained by the base constraint)
- rising and/or falling required times associated with each destination point (at least one must be specified)
- a boolean “min/max” attribute that says whether this is a min-delay constraint or a max-delay constraint
- other attributes which may be defined

It is important to note the the “other attributes” are a function of the base constraint and not of individual source or destination points. It may happen that the user wants some constraint attribute to take different values for some start or end points than for others. In that case, the base constraint must be split into two or more separate base constraints.

Constraint start and end points may be external ports of the design or pins of internal instances. Some tools (in particular, synthesis tools) may permit only certain instance pins to serve as constraint start and end points. For example, the Compass constraint language restricts constraint start and end points as follows:

A constraint start point may be:

- an external input port
- a sequential instance output port
- a black-box output port (note: in Compass terminology a “black box” is a leaf cell of the cell hierarchy for which no characterization of input-pin to output-pin delays is available)
- a broken-instance output pin (i.e. a pin associated with a user-disabled timing arc)

A constraint end point may be:

- an external output port
- a sequential instance input port
- a black-box input port
- a broken-instance input port

**Example 1.** Your design contains a clock that controls a number of edge-triggered storage elements. When you define a waveform for this clock, you are implicitly defining a max-delay constraint on your design. Described in words, this constraint states that a signal launched from one storage element on one clock cycle must arrive at any receiving storage element in time to be latched on the following clock cycle. This constraint meets the definition of a base constraint, because it can be described in terms of arrival times on the data-output pins of storage elements and required times on the data-input pins of storage elements.

**Example 2.** In the same design, you want to specify that a certain path from storage element A to storage element B is allowed to take two clock cycles rather than one. This is called a “multicycle constraint”. Let us also assume that there are other paths leaving storage element A which still must complete in one cycle, and other paths arriving at B which must complete in one cycle. On these assumptions, the multicycle constraint cannot be part of the same base

constraint as the original constraint, as different arrival or required times will be required on certain instance pins to express the multicycle requirement.

### 6.1.2 Primary Constraints and Timing Exceptions

A *primary constraint* is a constraint that is defined on its own and not derived from some other constraint. Other constraints are called “derived constraints”. A *derived constraint* is defined with reference to some parent constraint, which is a primary constraint.

Timing exceptions constitute one kind of derived constraints (in fact, the only kind considered in this specification).

A primary constraint is either a clock constraint or a combinational constraint. A *clock constraint* is automatically created when a clock or clock group is defined. (See the **gcl_clock** and **gcl_clock_group** commands.) A *combinational constraint* is defined independently of any clock. A combinational constraint can be created using either the **gcl_combinational_group** command or the **gcl_delay_path** command.

We can classify timing exceptions as follows:

a) point-to-point timing exceptions

   A point-to-point timing exception works by specifying some subset of the source points and some subset of the destination points of a specified parent constraint, and changing some part of the constraint definition just with respect to the “selected paths”. The selected paths are all timing paths that start at one of the specified source points and end at one of the specified destination points.

   Point-to-point exceptions include multicycle exceptions, disable exceptions, and path-delay override exceptions.

   A multicycle exception increases all required path delays for the selected paths by some fixed multiple of the period of the source or destination clock.

   A path-delay override exception sets the required path delay of all the selected paths to some specified value, overriding the required path delays imposed by the parent constraint.

   A disable exception specifies that the selected paths are not constrained by the parent constraint. It is equivalent to setting a point-to-point path-delay override of infinity.

b) internal-path timing exceptions

   An internal-path timing exception is similar to the point-to-point timing exception, except that the selected paths are defined in terms of “internal” points, that is, instance pins that are not necessarily constraint start or end points.

   The simplest case of the internal-path timing exception is the “internal-arc” timing exception, in which the selected set of paths consists of just a single timing arc of the parent constraint, that is, the path from an input pin to an output pin of one cell instance, or from an instance output pin to an instance input pin on the same net.

   Another form of internal-path timing exception is the “thru-points” timing exception, in which a set of paths is selected by giving a list of nets or instance pins and defining a path to be selected if it passes through all the points in the list.

   A still more general possibility is to allow each item in the thru-points list to be a wildcard or list of instance pins, with the requirement a selected path must contain at least one point for each item in the thru-points list. This is probably more generality than is needed, and also presents significant implementation problems; hence, this form of exception will not be included in this specification.

   We note that that Compass SET THRU command lets you specify a thru-points list having exactly two items, which may be wildcards. Thus, it is both more general (in allowing wildcards) and more limited (in allowing

only two items in the thru-points list) than the capability proposed here. Internal-path timing exceptions provide capabilities that are not available from point-to-point timing exceptions. For example, there may be two paths from start point A to end point Z, one of which includes an adder cell and the other of which includes a multiplier cell. The designer wishes to allow one or more extra clock cycles for the multiplier path, but not for the adder path. Thus, a multicycle exception needs to be defined just for the path that goes through the multiplier. Such an exception cannot be specified as a point-to-point exception, but can be provided by means of case-dependent constraints or internal-path timing exceptions.

### 6.1.3 Implementation of Timing Exceptions

In defining timing constraint commands, it is appropriate to consider the difficulty of implementation. If a certain form of constraint is of minimal usefulness and is difficult or impossible to implement efficiently, it should probably not be included in this specification. It is therefore helpful, when defining commands, to be guided by some notion of what practical algorithms are available to implement them.

One method of implementing point-to-point exceptions is by subdividing the parent constraint. For example, suppose the source points of a primary constraint consist of two subsets A and B, and its destination points consist of two subsets X and Y. Suppose that the user defines the set of paths from A to X to be a multicycle constraint. This can be accomplished by dividing the parent constraint into three base constraints as follows:

base constraint 1: A --> X (multicycle)

base constraint 2: A --> Y (single cycle)

base constraint 3: B --> X+Y (single cycle)

where X+Y is the union of X and Y.

If the user declares additional point-to-point exceptions, additional subdivisions can be performed. It is easy to imagine further optimizations of the process, to avoid excessive proliferation of base constraints.

The implementation of internal-path exceptions requires other techniques. If the internal path is a single arc, one can mark that arc in the database as being “broken” with respect to a particular constraint, and based on that information not propagate timing through that arc for that constraint.

More general internal-path exceptions require further algorithmic refinements, and raise efficiency issues. Before incorporating these more sophisticated capabilities into our specification, it would be wise to ask how useful they really are.

### 6.1.4 Other Variations

One possible variant way of defining exceptions is “inverse selection”. In this variation, instead of the exception applying to the selected path, it applies to all paths BUT the selected ones.

For example, inverse selection in a disable exception would mean that the constraint applies ONLY to the named paths, and all other paths are disabled.

The Compass SET THRU command is an example of an inverse-selecting internal-path exception.

It would be a simple matter to add an “inverse selection” option to all timing exception commands. However, in the absence of evidence that this option is offered as a general feature in any existing tool, we have decided to not include it in this specification.

### 6.1.5 Overlapping Constraints

Some synthesis tools allow the user to define overlapping constraints -- that is, to define more than one constraint affecting the same start or end point. The understanding is that these constraints are simultaneously enforced.

The ability to define overlapping constraints is seen as a desirable feature and thus is incorporated in this specification.

A consequence of overlapping constraints is that constraints should be nameable. That is, it must be possible to refer to a constraint other than by simply naming its endpoints.

### 6.1.6 Precedence of Timing Exceptions

What happens when the user defines more than one timing exception affecting the same start or end point?

Synopsys Design Compiler provides a set of precedence rules, specifying that certain timing exceptions take precedence over others, or that “more general” rules take precedence over more specific ones. For timing exceptions that have the same level of precedence, later commands override earlier ones.

For the present specification, we would like to provide compatibility with existing tools, but it is not desirable to tie the specification closely to one vendor by adopting an arbitrary set of rules that is specific to that vendor.

For this reason, we adopt the following rule:

> Whenever there is a potential conflict between two timing exceptions (i.e. there is some path whose treatment is specified differently by the two exceptions), the relative priority of the two exceptions must be explicitly indicated by means of an **-override** option.

This convention has the advantage of being less arbitrary than a fixed set of precedence relations. Moreover, it provides more flexibility (the user chooses the precedence) and at the same time can provide compatibility with a vendor’s fixed set of precedence rules.

The rule stated above is somewhat difficult to enforce, as it requires path tracing to see whether there are any conflicting paths. In practice, we allow a tool to enforce the following simpler rule: a conflict will be presumed to exist if both the start point list and the end point list of the two exceptions have nonempty intersection.

We have specified that exceptions are always defined with respect to some primary constraint. However, in many cases it will not be necessary to specify the parent constraint. When defining exceptions, the user has the following two choices:

- Specify a parent constraint (a primary constraint) for the exception. The start and end points for the new exception will be selected from the start and end points of the parent constraint.
- Do not specify a parent constraint. In this case, the exception will be applied to all previously defined constraints having nonempty intersection with both the start points and end points of the new exception. Thus, in this case, a single timing exception command can give rise to multiple exceptions, since a separate exception will be created for each primary constraint to which it is applicable.

One attractive feature of this system is “conservation of paths” -- the paths of a primary constraint are always accounted for. When first defined, a primary constraint has a certain set of paths. As exceptions are defined, some of these paths migrate to the exceptions. But at all times, each path resides in a unique constraint or exception.

### 6.1.7 Constraints and Path Groups

Constraints provide a natural basis for defining path groups.

A path group is a set of paths which the user wants grouped together for purposes of timing analysis or reporting. The possible uses of path groups include:

a) Definition of timing cost function. A typical approach is to assign a weight to each path group, and define the timing cost function as

   cost = SUM ( w(i)*cost(i) )

where w(i) is the weight of the i-th path group and cost(i) is the timing cost of the i-th path group. The timing cost of a path group can be defined in many ways -- for example, it could be defined as the worst negative slack associated with any path in the group.

b) Timing reports. The user may want the timing reports to be organized according to path group. For example, the timing report might report the worst slack for each path group.

Timing reports are not a concern of this specification. The question of whether we will define path group commands to control optimization is under discussion.

### 6.1.8 Broken Arcs

Disabled timing arcs are an important type of timing exception. The user may want to disable a timing arc in order to break a feedback loop or for other reasons.

It is natural to view a disabled timing arc not as an exception to some primary constraint, but rather as an exception to the default set of timing arcs as defined by the netlist and/or the cell library.

In terms of implementation, too, disabled timing arcs are likely to be treated differently from point-to-point timing exceptions. It is natural to assume that a timing-analysis tool has some internal representation of the timing connectivity of the netlist, in which each timing arc is represented as an arc in some directed acyclic graph or similar data structure. When the user specifies a disabled timing arc, the natural implementation is simply to remove the arc (or equivalently, flag it as “broken”) in the internal data base. This operation is independent of any timing constraint.

For this reason, we propose a separate command, **gcl_set_broken_arc**, to declare disabled timing arcs in the design.

### 6.1.9 Convention for Clock Constraint Start Points

Whenever a clock is defined, paths between sequential instances clocked by the clock are automatically constrained.

Logically, the start points of such constraints are the clock pins of the sequential instances. However, one often wants to refer to a path that goes through a specific output pin of a sequential instance. It is then convenient to use the name of the output pin to identify the start point of the path, while still understanding that the path really starts at the clock pin of the instance.

We therefore adopt the convention that, when a command requires as an argument a start point of an existing clock constraint, an output pin of a sequential instance may be named as the start point, with the effect of selecting paths that start at a clock pin of the instance and pass through the specfied output pin.

For example, suppose the sequential instance `u33` has clock pin CP and output pins Q and QN. The following identifiers may be used to refer to the start point of constrained paths originating at this instance:

`u33.CP` Selects all paths starting at `u33.CP` and passing through either the Q or QN output.

`u33.Q` Selects paths starting at `u33.CP` and passing through `u33.Q`.

`u33.QN` Selects paths starting at `u33.CP` and passing throught `u33.QN`.

In all these cases, one must remember that, although the sequential instance output pin is named as the start point of the constraint, the path actually starts at the clock pin, and the clock-to-output propagation delay through the cell is included in the delay of the selected paths.

This convention applies to clock constraints only. It does not apply to combinational constraints which happen to name a sequential instance output pin as a start point. For combinational constraints, the delay of the clock-to-output path can optionally be included in the path by using the **-include_se_delay** option of the **gcl_set_arrival_time** command.

### 6.1.10 Combinational Constraints

GCL offers two different ways to specify combinational constraints.

The *simple path delay* approach is convenient for specifying the delay of a combinational path with a specified start point and a specified end point. For example:

```
gcl_delay_path -max 10 -from IN1 -to OUT7
```

This command specifies that the delay of the slowest path from IN1 to OUT7 must be at most 10 time units.

The *arrival/required time* approach is a natural way to express combinational constraints involving multiple start or end points having differing arrival or required times. For example:

```
gcl_combinational_group cg1
gcl_set_arrival_time 0.4 IN1 -group cg1
gcl_set_arrival_time 1.3 IN2 -group cg1
gcl_set_arrival_time 0.6 IN3 -group cg1
gcl_set_required_time 10.0 OUT7 -group cg1
gcl_set_required_time 9.7 OUT8 -group cg1
```

The above commands constrain all paths that start at one of the ports IN1, IN2, IN3 and end at one of the ports OUT7 or OUT8. Note that each start point has a different arrival time and each output has a different required time. For example, IN1 arrives at time 0.4 and OUT7 is required at time 10.0. This means that the delay of the slowest path from IN1 to OUT7 must be at most 9.6 time units.

The **gcl_delay_path** command resembles the Synopsys **set_max_delay** command. Like the **set_max_delay** command, the **gcl_delay_path** command allows you to specify multiple start points and multiple end points. Moreover, analogously to the **set_max_delay** command, the **gcl_delay_path** command allows you to specify different arrival and required times for different start and end points, by using the **gcl_set_external_delay** command. For example, the above example using **gcl_combinational_group** can equivalently be expressed using the **gcl_delay_path** command as follows:

```
gcl_delay_path -max 10.0 -from { IN1 IN2 IN3 } -to { OUT7 OUT8 }
gcl_set_external_delay -input -max 0.4 IN1
gcl_set_external_delay -input -max 1.3 IN2
gcl_set_external_delay -input -max 0.6 IN3
gcl_set_external_delay -output -max 0.3 OUT8
```

This sequence of commands defines the very same set of constraints as the earlier command sequence starting with **gcl_combinational_delay**. The difference is in the way the individual arrival and required times are expressed. With **gcl_combinational_delay**, the arrival and required times are absolute times. With **gcl_delay_path**, one starts by defining a nominal delay for the path group (in this case, the nominal delay is 10 time units), and then one uses the **gcl_set_external_delay** command to define offsets from the nominal delay. Note that, for both start and end points, a positive external delay represents a shortening of the nominal delay. Thus, the external delay for OUT8 is 0.3, meaning that it is required 0.3 time units earlier than the nominal delay of 10 time units. The default offset is 0; thus, there is no need in the above example to specify the external delay of OUT7, because its external delay is 0, meaning that its required time is equal to the nominal delay of 10.

Various additonal features are provided with the **gcl_delay_path** command in order to provide compatibility with the various uses of the Synopsys **set_max_delay** and **set_min_delay** commands. For example, the **-constraint** and **-autoconstraint** options allow **gcl_delay_path** to be used to specify a point-to-point override of an existing clock constraint.

### 6.1.11 Design Guidelines for Constraint Commands

In light of the foregoing discussion, the following guidelines are adopted for specifying commands that define constraints:

1) We will provide commands for defining primary constraints and timing exceptions.
2) We will provide ways to change attributes of previously defined primary constraints and timing exceptions.
3) The user will be able to associate a name with each primary constraint or timing exception.
4) When two timing exceptions defined for the same constraint have some selected paths in common, the relative priority of the two exceptions must be explicitly indicated.
5) We will provide the ability to declare disabled timing arcs independently of any constraint.

## 6.2 CONSTRAINT COMMANDS

### 6.2.1 gcl_combinational_group

Syntax: **gcl_combinational_group** name

Declares a new combinational constraint group.

By default, there is one combinational constraint group, which has the name **default**. The default combinational constraint is available without being declared.

The commands **gcl_combinational_group** and **gcl_delay_path** offer two alternative methods for specifying combinational constraints.

See the examples given in the section “Combinational Constraints” above.

### 6.2.2 gcl_remove_combinational_group

Syntax: **gcl_remove_combinational_group** name

Removes a combinational constraint. All arrival and departure times associated with the constraint are cleared. The default combinational constraint (which has the name **default**) cannot be removed.

Arguments:

| | |
|---|---|
| name | Name of previously declared combinational constraint. |

Example:

```
gcl_remove_combinational_group cc2
```

### 6.2.3 gcl_set_arrival_time

Syntax: **gcl_set_arrival_time** value start_points [ **-max** | **-min** ] [ **-rise** | **-fall** ]
[ **-group** comb_group ]
[ **-include_clock_delay** ] [ **-include_se_delay** ]

Specifies arrival time of start points for a combinational constraint. The start points are most commonly external input ports, but may also be internal port instances.

For any specified start point that is a data output pin of a sequential instance, the option **-include_se_delay** causes the clock-to-output propagation delay through the instance to be subtracted from the given value, so that the specified value is in effect an arrival time for the signal at the clock pin of the instance.

For any specified source point that is a clock or data output pin of a sequential instance, the option **-include_clock_delay** causes the clock propagation delay (that is, the difference between the arrival time at the instance clock pin and the ideal clock edge) to be subtracted from the given value, so that the specified value reflects the clock propagation delay and skew.

A port instance may be a start point for more than one combinational constraint group, and may have a different arrival time specified for each constraint group.

Arguments:

| | |
|---|---|
| start_points | List of port instances. |
| value | Arrival time. |

Options:

| | |
|---|---|
| **-max** | Specifies maximum delay (used for setup check). |
| **-min** | Specifies minimum delay (used for hold check). |
| | If neither **-max** nor **-min** is specified, both max and min delay are set to the same value. |
| **-rise** | Indicates this value applies to rising signal only. |
| **-fall** | Indicates this value applies to falling signal only. |
| | If neither **-rise** nor **-fall** is specified, the value applies to both rising and falling directions of the signal. |
| **-group** comb_group | Name of combinational constraint group previously defined using the **gcl_combinational_group** command. If omitted, the default combinational constraint (i.e. **-group default**) is used. |
| **-include_clock_delay** | Subtract clock propagation delay from value, for any source point that is a clock or data output pin of a sequential instance. |
| **-include_se_delay** | Subtract clock-to-output propagation delay from value, for any source point that is a data output pin of a sequential instance. |

The initial default value of arrival time is “undefined”, which is to say the input is unconstrained.

Example:

```
gcl_set_arrival_time 1.0 A1 -rise -group cg1
gcl_set_arrival_time 1.4 BB[*] -fall -group cg1
```

The first command specifies that the rising edge of port `A1` arrives at time 1. The second command specifies that the falling edge of all ports matching the wildcard `BB[*]` arrives at time 1.4. These arrival times apply to both max and min delays.

### 6.2.4 gcl_set_required_time

Syntax: **gcl_set_required_time** value end_points [ **-max** | **-min** ] [ **-rise** | **-fall** ]
[ **-group** comb_group ]
[ **-include_clock_delay** ] [ **-include_setup_time** ]

Specifies required time of end points for a combinational constraint. The start points are most commonly external output ports, but may also be internal port instances.

For any specified source point that is a clock or data output pin of a sequential instance, the option **-include_clock_delay** causes the clock propagation delay (that is, the difference between the required time at the instance clock pin and the ideal clock edge) to be subtracted from the given value, so that the specified value reflects the clock propagation delay and skew.

A port instance may be a start point for more than one combinational constraint group, and may have a different required time specified for each constraint group.

Arguments:

| | |
|---|---|
| end_points | List of port instances. |
| value | required time. |

Options:

| | |
|---|---|
| **-max** | Specifies maximum delay (used for setup check). |
| **-min** | Specifies minimum delay (used for hold check). |
| | If neither **-max** nor **-min** is specified, the value applies just to max delay. |
| **-rise** | Indicates this value applies to rising signal only. |
| **-fall** | Indicates this value applies to falling signal only. |
| | If neither **-rise** nor **-fall** is specified, the value applies to both rising and falling directions of the signal. |
| **-group** comb_group | Name of combinational constraint group previously defined using the **gcl_combinational_group** command. If omitted, the default combinational constraint (i.e. **-group default**) is used. |
| **-include_clock_delay** | Add clock propagation delay to value, for any target point that is a data input pin of a sequential instance. |
| **-include_setup_time** | Subtract setup time fromvalue, for any target point that is a data input pin of a sequential instance. |

The initial default value of required time is “undefined”, which is to say the input is unconstrained.

Example:

```
gcl_set_required_time 10 A1 -rise -group cg1
```

The command specifies that the rising edge of port `A1` is required to arrive no later than at time 10. Because neither **-max** nor **-min** is specified, this applies to max delay checks only.

### 6.2.5 gcl_delay_path

Syntax: **gcl_delay_path** value [ **-max** | **-min** ] [ **-from** start_ports ] [ **-to** to_ports ]
[ **-constraint** cname | **-autoconstraint** ]

Defines a combinational constraint between specified start ports and specified end ports.

If **-constraint** and **-autoconstraint** are not specified, this command defines an independent combinational constraint. In this case, the start points are typically external input ports and the end points are typically external output ports; and the **-from** and **-to** options must not be omitted.

If **-constraint** or **-autoconstraint** is specified, the intention is to override specified paths of an existing clock constraint. In this case, the start and end points must be start and end point of that existing constraint. In this case, if **-from** is omitted, the start points are taken to be all start points of the overridden constraint; and if **-to** is omitted, the end points are taken to be all end points of the overridden constraint. It probably does not make sense for both **-from** and **-to** to be omitted, because then this command would completely override the existing clock constraint.

The arrival times of individual start points and required times of individual end points can be modified by subsequent **gcl_set_external_delay** commands.

This command gives much of the same functionality as the Synopsys **set_max_delay** and **set_min_delay** commands.

Arguments:

| | |
|---|---|
| from_ports | List of start points. |
| to_ports | List of end points. |
| value | Nominal path delay. |

Options:

| | |
|---|---|
| **-max** | Specifies maximum delay (used for setup check). |
| **-min** | Specifies minimum delay (used for hold check). |
| | If neither **-max** nor **-min** is specified, both max and min delay are set to the same value. |
| **-rise** | Indicates this value applies to rising signal only. |
| **-fall** | Indicates this value applies to falling signal only. |
| | If neither **-rise** nor **-fall** is specified, the value applies to both rising and falling directions of the signal. |
| **-constraint** cname | Name of existing clock constraint to be overridden with respect to the specified paths. The constraint is specified in the same way as the **-constraint** option of **gcl_exception_path**. |
| **-autoconstraint** | Override every existing clock constraint that has a start point in common with the **-from** list or a **-to** point in common with the -to list. |

Example:

See the examples in the section “Combinational Constraints” above.

### 6.2.6 gcl_clock

Syntax: **gcl_clock -waveform** waveform clock_ports

Declares one or more port instances of the design as clock insertion ports and associates a waveform with them.

By default the clock is an ideal clock with zero propagation delay. (See the **gcl_set_clock_delay** command.)

It is an error if any specified port instance is already a clock insertion port associated with a different waveform. If a specified port is already a clock insertion port with the same waveform, no action is taken.

Clock insertion ports can be used as start points of combinational constraints. But the clock's associated waveform does not constitute a declaration of arrival time for such a constraint. The **gcl_set_arrival_time** or **gcl_set_external_delay** command must be used to specify the arrival time.

Arguments:

clock_ports A list of port instances ("clock insertion ports") at which the waveform is to be inserted.

Example:

```
gcl_clock -waveform wave1 ck1
```

Declares `ck1` to be an clock with the waveform `wave1`.

### 6.2.7 gcl_remove_clock

Syntax: **gcl_remove_clock** clock_ports

Removes clock declaration from specified port instances. Any setting previously made by **gcl_set_clock_delay** either for one of the specified clock ports or for a port instance in the fanout of a specified clock port is cleared.

Arguments:

clock_ports List of port instances.

Example:

```
gcl_remove_clock ck1
```

Removes the clock declaration on the net `ck1`.

### 6.2.8 gcl_set_clock_delay

Syntax: **gcl_set_clock_delay** {**-propagated** | **-ideal** } [ **-delay** rf ] [ **-skew** sminmax ] ports

Specifies the method of computing clock delays for specified clock insertion ports or for specified sequential instance clock pins.

If the **-propagated** option is used, the **-delay** option must not be used, and any previous setting of **-delay** for the specified port instances is removed.

If any of the port instances has previously been assigned attributes of delay or skew, these attributes are changed according the the options present in the command, and any attributes not present in the command retain their previous values.

Arguments:

ports A list of port instances. Each port instance must be either a clock insertion port, or the clock pin of a sequential cell instance lying in the combinational fanout of a previously declared clock insertion port.

Options:

**-propagated** Indicates that the propagation from the root to each destination point should be computed based on the actual clock network that is present in the design. The **-skew** parameter may be used to add additional uncertainty. The **-delay** parameter is not used.

**-ideal** The clock is to be treated as an “ideal clock”. Its delay is to be computed from the values of the **-delay** and **-skew** parameters, ignoring any existing clock tree in the design.

**-delay** rf Propagation time for ideal clock. The argument rf is a list of one or two numbers.. If only one number is present, it defines both rising and falling delay. If two numbers are present, the first is rising delay and the second is falling delay. “Rising” and “falling” refer to the signal direction at the clock insertion port.

The default delay value is 0.

**-skew** sminmax Defines the uncertainty or variation in propagation time among destination points in the clock network. The argument sminmax is a list of one or two numbers. If two numbers are present, the first is to be subtracted from the propagation delay to get the smallest possible propagation time, and the second is to be added to the propagation delay to get the largest possible propagation time. If only one number is present, it is equivalent to a list of two items in which the number occurs twice.

“Skew” reflects the variation in propagation time within the fanout network of a clock insertion port. It does NOT reflect variations from the defined clock waveform between one clock pulse and the next, which is accounted for by the edge uncertainties in the waveform declaration (See the **gcl_waveform** command.) When comparing arrival times of clock signals having different parent waveforms, the edge uncertainty in the waveform must be considered in addition to the skew. Likewise, when comparing the arrival times of different clock edges associated with the same waveform, the edge uncertainty in the waveform must be considered. But for signals that originate from the same edge of the same cycle of the same parent waveform, the edge uncertainty of the waveform does not affect the difference of arrival times.

Example:

```
gcl_set_clock_delay ck2 -ideal -delay { .8 1.1 }
```

Changes `ck2` to be an ideal clock with propagation time .8 rising and 1.1 falling. The skew is not changed.

```
gcl_set_clock_delay ck3 -propagated -skew { 0 .2 }
```

Changes `ck3` to be a propagated clock, with a skew of .2 to be added to propagation time when determining maximum propagation delay, and no skew adjustment for minimum propagation delay.

```
gcl_set_clock_delay ck2 -ideal -delay 1.2 -skew .2
```

Changes ck2 to be an ideal clock with propagation delay 1.2 and skew .2. The skew is added to the propagation delay for maximum delay calculation, and subtracted from the propagation delay for minimum delay calculation.

```
/*
```

comment by tom schaefer:
Further possible refinements:
- allow four delay values instead of two, to cover the four cases of root rising or falling and destination pin rising or falling. I only defined two values because that is what Synopsys and Compass do.
- allow different min and max skew values to be defined for each of the four possible delay cases.

```
*/
```

### 6.2.9 gcl_unset_clock_delay

Syntax: **gcl_unset_clock_delay** ports

Removes the clock delay setting from the specified clock insertion ports or sequential instance clock pins.

When the clock delay setting is removed from a sequential instance clock pin, its delay treatment is governed by the clock delay setting of the clock insertion port from which the clock signal originates.

When the clock delay setting is removed from a clock insertion port, it becomes an ideal clock with zero delay and zero skew.

Arguments:

| | |
|---|---|
| ports | List of port instances. |

Example:

```
gcl_unset_clock_delay ck1
```

Removes the on the clock delay setting from the clock insertion port `ck1`. It becomes an ideal clock with zero delay and zero skew.

### 6.2.10 gcl_clock_group

Syntax: **gcl_clock_group** [ **-add** | **-delete** ] name waveforms

Defines a set of waveforms constituting a common clock domain.

Arguments:

| | |
|---|---|
| name | Name for the clock group. Must be a new name unless **-add** or **-delete** is used. |
| waveforms | List of names of previously defined waveforms. |

Options:

| | |
|---|---|
| **-add** | Add waveforms to a previously declared clock group. |
| **-delete** | Delete waveforms from a previously declared clock group. |

Example:

```
gcl_clock_group cg1 ck1 ck2
```

Declares a clockgroup consisting of waveforms `ck1` and `ck2`.

### 6.2.11 gcl_remove_clock_group

Syntax: **gcl_remove_clock_group** name

Removes an existing clock group declaration.

Arguments:

| | |
|---|---|
| name | Name of clock group. |

Example:

```
gcl_remove_clock_group cg1
```

After this command, the clock group `cg1` no longer exists.

### 6.2.12 gcl_set_external_delay

Syntax: **gcl_set_external_delay** { **-input** | **-output** } value ports [ **-max** | **-min** ] [ **-rise** | **-fall** ]
[ **-posedge** wave | **-negedge** wave ]

For clock constraints, specifies the external input or output delay of specified ports relative to a specified clock edge. For combinational constraints declared with **gcl_delay_path**, specifies reduction in nominal path delay for paths having the specified start or end points.

If **-posedge** or **-negedge** is specified, this command refers to a clock constraint, and the specified ports are typically external ports having a path to or from one or more sequential instances clocked by the specified waveform.

If neither **-posedge** nor **-negedge** is specified, this command refers to combinational constraints defined by previous **gcl_delay_path** commands. The specified port must be ports that were named as **-from** or **-to** ports in one or more previous **gcl_delay_path** commands.

Arguments:

| | |
|---|---|
| source_points | List of port instances. Typically these are external input ports if the **-input** option is used, and external output ports if the **-output** option is used. |
| **-input** | The specified port instances are start points of the clock constraint. |
| **-output** | The specified port instances are end points of the clock constraint. |
| value | The interpretation of the value depends on the other options:<br>If **-posedge** or **-negedge** is specified:<br>With the **-input** option: amount of time after the reference clock edge when the signal arrives at the start points.<br>With the **-output** and **-max** options: amount of time before the reference clock edge when the signal must be available.<br>With the **-output** and **-min** options: amount of time before the reference clockedge, earlier than which the signal must not change.<br>If neither **-posedge** nor **-negedge** is specified:<br>Amount of time by which the nominal path delay specified in any previous **gcl_delay_path** command is reduced for paths having one of the specified start or end points. |

Options:

| | |
|---|---|
| **-max** | Specifies maximum delay (used for setup check). |
| **-min** | Specifies minimum delay (used for hold check).<br>If neither **-max** nor **-min** is specified: for the **-input** case, the delay value applies to both max and min delays; for the **-output** case, the delay value applies just to the max case. |
| **-rise** | Indicates this value applies to rising signal only. |
| **-fall** | Indicates this value applies to falling signal only. |

If neither **-rise** nor **-fall** is specified, the value applies to both rising and falling directions of the signal.

**-posedge** wave — External delay is relative to the rising edge of the waveform wave.

**-negedge** wave — External delay is relative to the falling edge of the waveform wave.

The initial default value of arrival time is “undefined”, which is to say the signal is unconstrained.

Example:

```
gcl_set_external_delay -input 1.0 * -rise -posedge ck1
gcl_set_external_delay -input 1.4 * -fall -posedge ck1
```

The first command specifies that the rising edge of all external input ports arrives 1 time unit after the rising edge of waveform `ck1`. The second command specifies that the falling edge of all external input ports arrives 1.4 time units after the rising edge of waveform `ck1`. These arrival times apply to both setup and hold checks.

```
gcl_set_external_delay -output -max 2.0 ZZY -negedge ck1
gcl_set_external_delay -output -min -3.2 ZZY -negedge ck1
```

The first command specifies that the external output ZZY must arrive no later than 2.0 time units before the falling edge of ck1. The second command specifies that ZZY must not change earlier than 3.2 time units after the falling edge of ck1. Note the use of negative value (-3.2) to refer to time after the clock edge. Note also that the reference clock edge for the second command is normally one cycle earlier than the reference clock edge for the first command (see the discussion of reference clock edges in the GCF specification).

See also the combinational examples given in the section “Combinational Constraints” above.

### 6.2.13 gcl_unset_external_delay

Syntax: **gcl_unset_external_delay** { **-input** | **-output** } ports [ **-max** | **-min** ] [ **-rise** | **-fall** ]
[ **-waveform** wave ]

Removes one or more external delay settings.

Arguments:

ports — List of ports.

Options:

**-waveform** wavename — Name of waveform.

For other options, see **gcl_set_external_delay**. Only external delays corresponding to the specified options are unset.

Example:

```
gcl_unset_external_delay -input in1
```

All arrival times associated with input port `in1` are cleared.

### 6.2.14 gcl_exception_path

Syntax: **gcl_exception_path** [ **-name** name ]
[ **-constraint** waves | **-group** comb_group ]
{**-add_source_cycles** n | **-add_target_cycles** n | **-add_delay** rf | **-unconstrain** | **-revert** }

[ **-setup** | **-hold** ]
[ **-from** startpoints | **-from_instance** startinsts ]
[ **-to** endpoints | **-to_instance** endinsts ]
[ **-from_rise** | **-from_fall** ] [ **-to_rise** | **-to_fall** ]
[ **-thru** thruPoints ]
[ **-override** excepts ]

Creates a path-based timing exception to a primary constraint, or changes attributes of a previously created exception.

The primary constraint is either a clock constraint or a combinational constraint.

The keyword **-add_source_cycles**, **-add_target_cycles**, **-add_delay**, **-unconstrain** or **-revert** indicates which kind of exception is being defined.

The options **-add_source_cycles**, **-add_target_cycles** and **-add_delay** all add a specified amount of time to the required path delays of a specified set of paths. The specified amount of time represents an addition to the default required delays of the parent constraint, or a reduction in the required delays if the option argument is negative.

The option **-unconstrain** causes the specified paths to be unconstrained. Conceptually, this is equivalent to **-add_delay** n with n equal to infinity.

The option **-revert** removes the timing exception with respect to the specified paths and restores their delay requirement to that of the parent constraint.

If the named exception already exists, this command changes attributes of the previously defined exception. The attributes that can be changed are **-add_source_cycles**, **-add_target_cycles**, **-add_delay**, **-delay**. Any of these keywords may be used to change a value for itself or any other of these keywords. The new setting is always specified relative to the default required delays of the parent constraint.

The start points and end points specified for the timing exception must be start and end points of the parent constraint. Wildcards used in specifying start and end points match only port instances that are start or end points of the parent constraint. To override the timing on a path that starts at a point other than a constraint start point or ends at a point other than a constraint end point, a combinational constraint should be used.

Options:

**-name** name — Optional name for this exception. Required if this constraint is to be overridden by another constraint.

**-constraint** waves — Parent clock constraint. The argument waves is a list of one or two waveform names. If one waveform is named, the parent constraint is the constraint defined by the single named waveform. If two waveforms are named, they must belong to the same clock domain (see **gcl_clock_group**). The exception then affects paths having the first waveform as source waveform and the second waveform as target waveform.

**-group** comb_group — Name of parent combinational constraint group.

If both **-constraint** and **-group** are omitted, the exception will apply to every primary constraint that has nonempty intersection with the specified startpoint and endpoint lists.

**-add_source_cycles** n — Increase the required delay of all specified paths by n periods of the source clock. Not permitted with a combinational parent constraint.

**-add_target_cycles** n — Increase the required delay of all specified paths by n periods of the target clock. Not permitted with a combinational parent constraint.

**-add_delay** rf — Increase the required delay of all specified paths by specified number of time units. The argument rf is a number or a list of two numbers. If two numbers the first applies to the

rising signal direction at the endpoint and the second to the falling direction. If one number, it applies to both rising and falling signals.

**-unconstrain** Make the specified paths unconstrained by the parent constraint.

**-revert** Restore the delay requirement of the parent constraint to the specified paths.

**-setup** Exception applies only to setup checks. (This is the default.)

**-hold** Exception applies only to hold checks.

**-from_rise** Exception applies only to paths with rising signal at start point.

**-from_fall** Exception applies only to paths with falling signal at start point.

**-to_rise** Exception applies only to paths with rising signal at end point.

**-to_fall** Exception applies only to paths with falling signal at end point.

**-from** startpoints List of port instances that are start points of the parent constraint. Wildcards used here match only such port instances.

**-from_instance** startinsts List of start instances. List items are instance names. Each instance name selects all ports of the instance that are start points of the parent constraint.

**-to** endpoints List of port instances that are end points of the parent constraint. Wildcards used here match only such port instances.

**-to_instance** endinsts List of end instances. List items are instance names. Each instance name selects all ports of the instance that are end points of the parent constraint.

**-thru** thrupoints List of thru points. Only timing paths that pass through every point in the list are disabled. Each list item is a port instance.

**-override** excepts List of names of timing exceptions that are overridden by this one. Must be the names of previously created timing exceptions. This option is transitive (if A overrides B and B overrides C, then A automatically overrides C). Cyclic overrides are prohibited (for example, if A overrides B and B overrides C, then C must not overide A).

Examples:

```
gcl_exception_path -name mc1 -constraint ck1 -add_target_cycles 1
-from_instance U1.* -to_instance U2.*
```

Declares a multicycle exception. All paths of constraint `ck1` that start at a pin of an instance whose name matches `U1.*` and end at a pin of an instance whose name matches `U2.*` have their default required delay increased by one cycle of the target clock, compared to the default required delay imposed by the parent constraint. Since the **-hold** option is not used, the default clock edge for hold checking remains the edge that occurs one cycle before the latching edge.

```
gcl_exception_path -name dis1 -constraint ck1 -unconstrain -from int1
```

Unconstrains all timing paths of the constraint `ck1` that start at the external input `int1`.

/*

comment by tom schaefer
I was at first considering having a **-delay** option (as an alternative to **-add_delay**) in this command, which would have specfied an absolute (as opposed to incremental) delay for the selected paths. But it started to look like a can of worms defining how precisely the delay of the path should be calculated (I would have needed to offer the same variations that are offered by the **-include_clock_delay** and **-include_setup_time** options of **gcl_set_required_time** and it seemed like a bit too much complexity for one command. If you want to specify an absolute delay, you should use a combinational constraint.

The intended use of the **-hold** option of this command is not extremely clear. The effect of this options is to suppress hold checking of an immediately preceding clock edge, without any specific indication of why it is okay to do so. The only reason I know of is that the preceding clock edge is suppressed by clock gating: if that is the case I would much prefer to have a command that directly asserts the omission of the clock edge, since this information could be in principle be exploited to perform more accurate timing analysis.

```
*/
```

### 6.2.15 gcl_remove_exception_path

Syntax: **gcl_remove_exception_path** name

Removes a timing exception previously declared with **gcl_exception_path**. Paths associated with the exception revert to the parent constraint.

Arguments:

| | |
|---|---|
| name | Name of exception to be removed. |

Example:

```
remove_exception_path dis1
```

### 6.2.16 gcl_set_broken_arc

Syntax: **gcl_set_broken_arc** [ **-rise** | **-fall** ] { [ **-from** frompoints ] [ **-to** topoints ] | [ **-instance** insts | **-net** nets ] }

Marks timing arcs as disabled.

Delays will not be propagated along broken arcs.

The timing arcs may be “intra-cell” timing arcs or “intra-net” timing arcs. An intra-cell timing arc starts at an input pin of a leaf cell instance and ends at an output pin of the same cell. An intra-net timing arc stars at a “driving” port instance and ends at a “receiving” port instance on the same net. A “leaf cell instance” is an instance whose timing is described as a primitive cell and which is not viewed as containing other instances).

If **-from** and **-to** are both specified, this command disables all timing arcs which start at one of the frompoints and end at one of the topoints.

If only one of **-from** and **-to** is specified, all timing arcs ending or starting at the specified port instances are disabled.

If **-instance** is specified, all intra-cell timing paths of the specified leaf cell instances are disabled, subject to the **-rise** and **-fall** options.

If **-net** is specified, all intra-net timing paths of the specified nets are disabled, subject to the **-rise** and **-fall** options.

Options:

| | |
|---|---|
| **-from** frompoints | List of start-of-arc port instances. |
| **-to** topoints | List of end-of-arc port instances. |
| **-instance** insts | List of instances, through which all timing paths are to be disabled. This is equivalent to using **-from** with a list of all input pins of the instances and **-to** with a list of all output pins of the instances. |
| **-net** nets | List of nets which are to be completely broken. |
| **-rise** | Disable only the rising signal at the arc end point. |

**-fall** Disable only the falling signal at the arc end point.

Example:

```
gcl_set_broken_arc -rise -from u1.d -to u1.q
```

Disables the D-to-Q path of instance u1. Only the rising output at Q is disabled. The path from D to Q falling is not affected.

```
gcl_set_broken_arc -to U1.*
```

Assuming that U1 is a leaf cell instance of the design, this command disables all timing paths through the instance U1.

### 6.2.17 gcl_unset_broken_arc

Syntax: **gcl_unset_broken_arc** [ **-from** point1 ] [ **-to** point2 ] [ **-all** ]

Enables a timing arc that was previously disabled.

Some tools may disable certain timing arcs by default. This command can be used to enable those arcs, if the tool permits. To do this, the **-from** or **-to** option must be used.

Options:

**-from** Port instance at which arc begins.

**-to** Port instance at which arc ends.

**-all** Unset all user-set broken arcs. This option is valid only if **-from** and **-to** are not used.

Example:

```
gcl_unset_broken_arc -to *.Q *.QN
```

Enables all timing arcs that end at the Q or QN pin of any instance in the design. (Depending on the tool, this could have the effect of causing clock-to-Q and reset-to-Q paths to be treated as combinational paths.)

```
gcl_unset_broken_arc -all
```

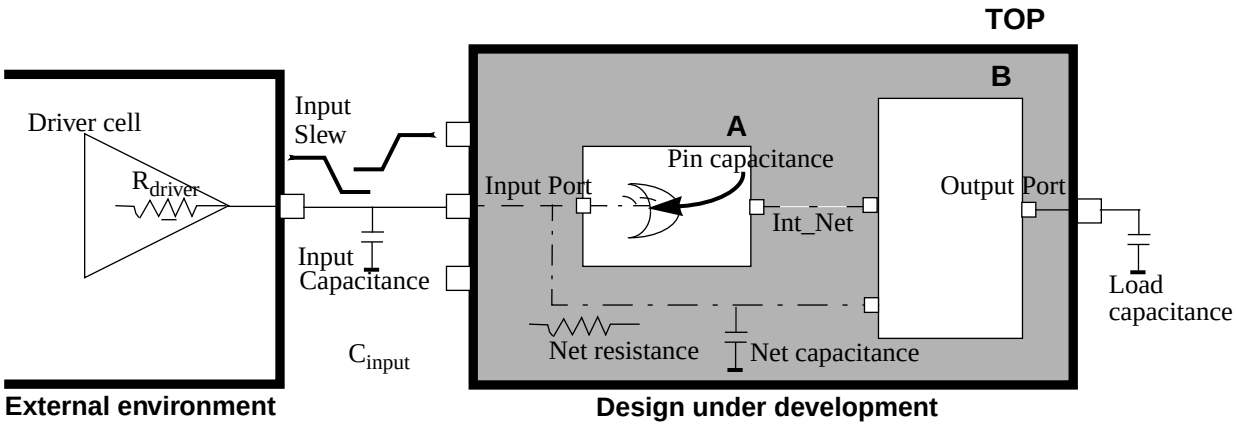
Enables all timing arcs that have previously been disabled by the **gcl_set_broken_arc** command, but does not enable timing arcs that have been broken by default by the tool.

# Section 7

# Parasitics Commands

## 7.1 Discussion of Parasitics specification

**Figure 1-1Modelling internal and external parasitics**




The model shown in Figure 1-1 is used to describe the parasitic environment for the design. The constraint commands listed in this section will allow specification and reporting of the various constraint elements illustrated. For input ports, driver resistance(rise, fall) will be used alongwith input capacitance to arrive at input slew specification for rise, fall.

## 7.2 Constraint commands

### 7.2.1 gcl_set_load

/* Jay's comment: I did not specify set_external_load, set_internal_load, load explicitly, eventhough I see them specified separately in the GCF spec. In my experience, one single command with port/net activity scope has sufficed.

Also, what is considered "external" at a block level, will become "internal" at a (higher) design level.

I did not specify automatic wild-carding for this constraint - "If no port/net is specified, the specification applies to all".

*/

Syntax: **gcl_set_load** capacitance ports_or_nets

Specifies the value of load attribute to be set for the specified ports or nets. The specified ports or nets could be internal or external to the design.

Arguments:

| | |
|---|---|
| capacitance | value of capacitance to be set on listed objects. |
| ports_or_nets | list of port names or net names. |

Options:

none

Example:

```
gcl_set_load 2.3 TOP/A/port1
```

specifies that the 2.3 units of capacitance are to be set to the load attribute for “TOP/A/port1”.

### 7.2.2 gcl_set_fanout

Syntax: **gcl_set_fanout** num_loads port_list

Specifies the capacitance in terms of number of loads for specified ports. This constraint may be specified for both input and output ports. If no port is specified, the specification applies to all primary ports. The current wire load model will be used for translating the “number of loads” into capacitance.

Arguments:

| | |
|---|---|
| num_loads | numerical value of load |
| port_list | list of port names |

Options:

none

Example:

```
gcl_set_fanout 1.5 TOP/A/port1
```

specifies that the 1.5 loads are to be set to the load attribute for “TOP/A/port1”. A current wire load must be available to convert this into capacitance for timing calculation.

### 7.2.3 gcl_set_resistance

/* Jay’s note: I know that SPF should be used for specifying detailed RC tree information for interonnects. However, atleast for some more time, we will have the older simple R,C back-annotation flow in EDA tools. To allow use of existing tools, I would recommend inclusion of this “lumped parameter” construct.

*/

Syntax: **gcl_set_resistance** resistance netname_list

Specifies the value of load attribute to be set for the specified ports or nets. The specified ports or nets could be internal or external to the design.

Arguments:

| | |
|---|---|
| resistance | value of resistance to be set on listed objects |
| netname_list | list of net names |

Options:

none

Example:

```
gcl_set_resistance 9.4 TOP/B/net35
```

specifies that the 9.4 units of resistance are to be set to the resistance attribute for net “TOP/B/net35”.

## 7.3 Related commands

### 7.3.1 gcl_set_driver_resistance

Syntax: **gcl_set_driver_resistance** resistance [-rise] [-fall] port_list

Specifies the output resistance for the driver cell that drives the specified port. This command is used to model the external environment. If -rise, -fall are not specified, then the resistance value is applied to both cases.

Arguments:

| | |
|---|---|
| resistance | value of resistance to be set for drivers of specified ports |
| port_list | list of port names |

Options:

| | |
|---|---|
| -rise | specifies rise resistance |
| -fall | specifies fall resistance |

Example:

```
gcl_set_driver_resistance 4.8 -rise port15
```

### 7.3.2 gcl_set_max_transition

Syntax: **gcl_set_max_transition** value [-rise] [-fall] port_or_design

Specifies the maximum transition time for the specified port. If -rise, -fall are not specified, then the transition value is applied to both cases.

Arguments:

| | |
|---|---|
| value | value of maximum transition time to be set for drivers of specified ports or design |
| port_or_design_names | list of port names or design names |

Options:

| | |
|---|---|
| -rise | specifies rise transition time |
| -fall | specifies fall transition time |

Example:

```
gcl_set_max_transition 5.1 -fall port15
```

### 7.3.3 gcl_set_max_capacitance

Syntax: **gcl_set_max_capacitance** value port_or_design

Specifies the maximum capacitance value for the specified port or design.

Arguments:

value value of maximum capacitance to be set for drivers of specified ports or design

port_or_design_names list of port names or design names

Options:

None

Example:

```
gcl_set_max_capacitance 3.9 {port18 designB}
```

# Section 8

# Area Commands

## 8.1 Constraint commands:

The area constraint commands allow specification for primitive area as well as total area. Primitive area is defined as the cummulative area of the leaf level primitive cells which are instantiated either directly within a cell or within its descendents. This area does not include various types of physical overhead such as routing and power distribution which affect the total area of the cell.

The total area on the other hand specifies a constraint on the total area of the cell, including all physical overheads.

/* Raghu's note: I see 2 keywords in the GCF spec, primitive_area and total_area and both of them specify one value but could optionally specify 2 values. No other command in GCL allows 2 values to be specified. So I am defining separate commands for the min and max of these 2 values. That could be a good approach also, since it is consistent with all the other commands */

/* Also very important! There is no scope specification avaliable in GCL yet. So there is no way to know which cell these commands apply to! We could have a -cell option added to these commands. But, I think it is important to discuss this. If there is a scope specification avaliable then the -cell option is not necessary. Also, we need a scope specification for all the other commands as well; power, timing and parasitics. */

### 8.1.1 gcl_set_min_primitive_area

Syntax: **gcl_set_min_primitive_area** value

Specifies the minimum cummulative primitive area of the leaf level primitive cells which are instantiated either directly with a cell or within its descendents. This area does not include various types of physical overhead such as routing and power distribution which affect the total area of the cell.

Arguments:

value - The minimum primitive area constraint for this cell.

Options:

None.

Example:

gcl_set_min_primitive_area 500

### 8.1.2 gcl_set_max_primitive_area

Syntax: **gcl_set_max_primitive_area** value

Specifies the maximum cummulative primitive area of the leaf level primitive cells which are instantiated either directly with a cell or within its descendents. This area does not include various types of physical overhead such as routing and power distribution which affect the total area of the cell.

Arguments:

value - The maximum primitive area constraint for this cell.

Options:

None.

Example:

gcl_set_max_primitive_area 15000

### 8.1.3 gcl_set_min_total_area

Syntax: **gcl_set_min_total_area** value

Specifies the minimum total area of the cell including physical overhead.

Arguments:

value - The minimum total area constraint for this cell.

Options:

None.

Example:

gcl_set_min_total_area 5000

### 8.1.4 gcl_set_max_total_area

Syntax: **gcl_set_max_total_area** value

Specifies the maximum total area of the cell, including physical overhead.

Arguments:

Value - The maximum total area constraint for this cell.

Options:

None.

Example:

gcl_set_max_total_area 15000

### 8.1.5 gcl_set_max_porosity

The porosity of a cell may be specified using the porosity constraints. Porosity is a measure of how much space is available for over-the-cell routing. It is defined as the percentage of the cummulative primitive area which is available for over-the-cell routing. The cummulative 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: **gcl_set_max_porosity** value

Specifies the maximum porosity of the cell.

Arguments:

Value - The maximum porosity constraint for this cell.

Options:

None.

Example:

gcl_set_max_porosity 40

This sets the maximum porosity of the cell to 40% of the cummulative primitive area of the cell, which indicates that atleast 40% of the cummulative primitive area is available for over-the-cell routing.

### 8.1.6 gcl_set_min_porosity

Syntax: **gcl_set_min_porosity** value

Specifies the minimum porosity of the cell.

Arguments:

Value - The minimum porosity constraint for this cell.

Options:

None.

Example:

gcl_set_min_porosity 20

This sets the minimum porosity of the cell to 20% of the cummulative primitive area of the cell, which indicates that upto 20% of the cummulative primitive area is available for over-the-cell routing.

# Section 9

# Power Commands

## 9.1 Power Commands

The commands in this section describe constraints on the average power consumed by the cell and the primitives instantiated within it, as well as on the power consumed by particular nets.

### 9.1.1 gcl_set_avg_cell_power

Syntax: **gcl_set_avg_cell_power** value [-**min**] [-**max**] [-**case** case_name]

Specifies the average power which may be consumed by the current instance. If -min, -max are not specified, then the value is applied to both cases.

Arguments:

value value of average power to be set for the current instance

port_or_design_names list of port names or design names

Options:

-**min** specifies minimum power value

-**max** specifies maximum power value

-**case** specifies the name of the case in which the constraint applies

Example:

```
gcl_set_avg_cell_power 50 -max
```

### 9.1.2 gcl_set_avg_net_power

Syntax: **gcl_set_avg_net_power** value [-**min**] [-**max**] [-**case** case_name] port_names

Specifies the average power which may be consumed by a net (typically a clock net). The net is specified using one of the ports connected to the net. If -min, -max are not specified, then the value is applied to both cases.

Arguments:

value value of average power to be set for each of the nets connected to one of the ports

port_names list of port names

Options:

-**min** specifies minimum power value

-**max** specifies maximum power value

-**case** specifies the name of the case in which the constraint applies

Example:

```
gcl_set_avg_net_power 20 -max clk1 clk2
```

# Section 10

# Discussion of Issues

This final section is reserved for informal discussion of various issues in the specification of GCL. This section is seen as temporary in nature. Material in this section will eventually either be incorporated into the main document in a more finished form, or will be removed from the document.

## 10.1 Features Not Yet Addressed

The following features are offered in the GCF specification, but have not yet been addressed in the context of GCL:

a) CASE construct : We need to define a mechanism for specifying case-dependent constraints.

b) LEVEL construct : Commands and features are described here without regard to the GCF level of each construct.

c) subdesign constraints: By what sequence of commands does the user specify that a contraint applies to a subdesign rather than to the whole top-level design?

## 10.2 SET vs CREATE

In the COMPASS command language, the **set** command is used loosely to cover a wide range of uses:

- setting an attribute of a design object (**set load**)
- creating an analysis entity (**set clock**)
- giving a directive to the compiler (**set cpueffort**)

One is tempted to try to come up with some more precise command names. For example, Synopsys has “create_clock”, “create_wire_load”, etc.

What really is the difference between **set** and **create**? As a working definition, we could say that **create** creates a whole new design object, obliterating any previous attribute settings, whereas the **set** command can be used to override attribute settings.

With this definition, do we really want to have a “create clock” command? Notice that the Synopsys create_clock command not only creates a design entiry (a clock waveform), but also sets some of its attributes (e.g. its period).

It is very desirable to be able to change the attributes of a clock (for example, change its period) without destroying the existing clock. Thus, in terms of our working definition, it is desirable to have a “set clock_attribute” command. But then the “create clock” command is really redundant, because it is a simple matter to have the “set clock_attribute” command create the clock if it does not exist.

Thus, in this proposal, the command corresponding to Synopsys create_clock is “set waveform” and not “create waveform”. This command not only sets the attributes of a waveform, but creates the waveform if it does not exist.

It seems as if the above sort of considerations may make it completely unnecessary to have a CREATE command.

SHOW and UNSET

In the COMPASS command language, every SET command has a corresponding SHOW command and CLEAR command. Synopsys seems to use “report_xxx” instead of “show xxx” and “remove_xxx” instead of “clear xxx”.

In this spec, I am proposing to use UNSET as the opposite of SET, and REMOVE as the opposite of CREATE.

- **unset_xxx** : remove the attribute value that was assigned by “**set_xxx**”. The attribute then reverts to its default value, if it has a default value, or simply to the “unset” state if it has an an initial state distinct from any assigned value.
- **remove_xxx** : removes a design object (opposite of “create”)

## 10.3 Override Conventions

This section defines the effect of giving two or more commands that attempt to set the same design object or attribute.

For commands that set the value of an attribute of some design or design object, the rule is that the latest command overrides all previous ones.

The override affects just the specific attribute or design object referred to by the overriding command. Thus, a command using a wildcard name can be followed by a command using a more specific name, with the effect that the second command overrides just with respect to the more specific name.

For example:

```
gcl_set_load .6 allOutputs()
gcl_set_load .5 z*
gcl_set_load .3 z2
```

If the design has outputs named y1, z1 and z2, the effect of the above sequence of commands is to set the loads as follows:

| | |
|---|---|
| y | 1 .6 |
| z | 1 .5 |
| z2 | .3 |

For commands that control multiple attributes, the override applies to the particular attribute that is assigned, while leaving previously set attributes unchanged. For example:

```
gcl_set_drive 2.0 a*
gcl_set_drive 1.0 a1*
gcl_set_drive -rising .8 a15 a25
```

If the design has inputs named a5, a14, a15 and a25, the above sequence of commands set their drives as follows:

| | rising drive | falling drive |
|---|---|---|
| a5 | 2.0 | 2.0 |
| a14 | 1.0 | 1.0 |
| a15 | .8 | 1.0 |
| a25 | .8 | 2.0 |

In the event that a design object has multiple synonymous names, it must be remembered that a wildcard can match any of these names. Thus, in the last example, if “a15a” was a synonym of “a5”, the drive value of this input would end up as 1.0 instead of 2.0.

For commands that do not set an attribute value, the effect of overrides will be described on a case by case basis.

## 10.4 Syntax for lists

For ease of command entry, I really like the convention that a list is just a series of identifiers, separated by blank space -- no special delimiters. Thus, if we wanted to set the arrival times of both int1 and int2, the above command would become

```
gcl_set_external_delay -input 3.0 int1 int2 -max -posedge ck1
```

The questions this raises are: (1) Is it unambiguous? and (2) Is parsing easy enough?

Ambiguous? - Yes it is ambiguous. If we decide to put a command argument that is a list at the end of the command, it cannot be distinguished from an option argument that is a list. For example:

```
gcl_set_clock -delay 1.0 1.3 ck1 ck2
```

1.0 and 1.3 are rising and falling delay, and you see we have a bit of a problem distinguishing them from the list of command arguments “ck1 ck2”.

So we need some more rules. Here are some alternatives that I see:

a) require all command arguments to precede all option

```
gcl_set_clock ck1 ck2 -delay 1.0 1.3
```

b) use some kind of delimiter to delineate any list that contains more than one item. for example, we could use curly braces as a list delimiter:

```
gcl_set_clock -delay { 1.0 1.3 } { ck1 ck2 }
```

c) carrying this a bit further, require list delimiters for ALL list arguments, even if they are only one item. thus, braces would be mandatory in the following:

```
gcl_set_clock { ck1 } -delay { 1.0 }
```

I think that (c) is rather annoying, assuming that people are actually going to be typing these things -- although it does make for easy parsing. This rule also lacks flexibility: for example, suppose we had originally defined **-delay** to be a scalar, and later decided to make it a list because we need to have both rising and falling delay. Under rule (c) any previously written command files would have illegal syntax.

I have written the command descriptions in accord with (a), mainly because it was a little easier to write the descriptions that way. But I am certainly open to (b).

### 10.4.1 CREATE vs SET (further comments)

For the most recent GCL draft, I decided to observe the distinction between “set” and “create”, only without using the word “create”.

**set** is used for commands that set or change some attribute of an existing object. Its action is reversed by **unset**.

**gcl_set_external_delay** sets attribute of external delay on port instances. **gcl_set_clock** sets attributes of waveform name, delay, and skew on external input ports

If a command *creates* some named object or significant data structure that did not exist before, I do not use **set**. Instead, I just use the single-word command by itself.

Example: **gcl_multicycle_path** creates a multicycle exception. The user can give a name to the exception, so that it can be referred to in a subsequent command. **gcl_waveform** creates a waveform, which has a name assigned by the user. **gcl_clock_group** creates a clock group, which has a name assigned by the user.

Whenever the user is able to assign a name to the new entity created by a command, the command should be considered a “create” command. In the case of the **gcl_clock** command, the user does not get to assign a new name to the clock, but this is still regarded as a “create” command because there are normally data structures associated with a clock insertion port that go beyond the simple assigning of attribute values.

I did not actually use the word “create” in the command name, because one of the things these commands can do is modify the attributes of a previously created object. For example, the **gcl_multicycle_path** command has a **-change** option that lets you change the value of the multiplier of an existing multicycle exception. And the **gcl_clock_group** command has **-add** and **-delete** options that add and delete items to an existing clock group.

I use REMOVE to reverse the effect of a “create” command. For example, **gcl_remove_multicycle_path** removes the multicycle exception, so it no longer exists as a named entity. (An alternative would have been “**gcl_multicycle -remove**” but I felt this was too error-prone: for example, someone could accidentally type “**gcl_clockgroup -remove**” when they meant “**gcl_clockgroup -delete**”.)