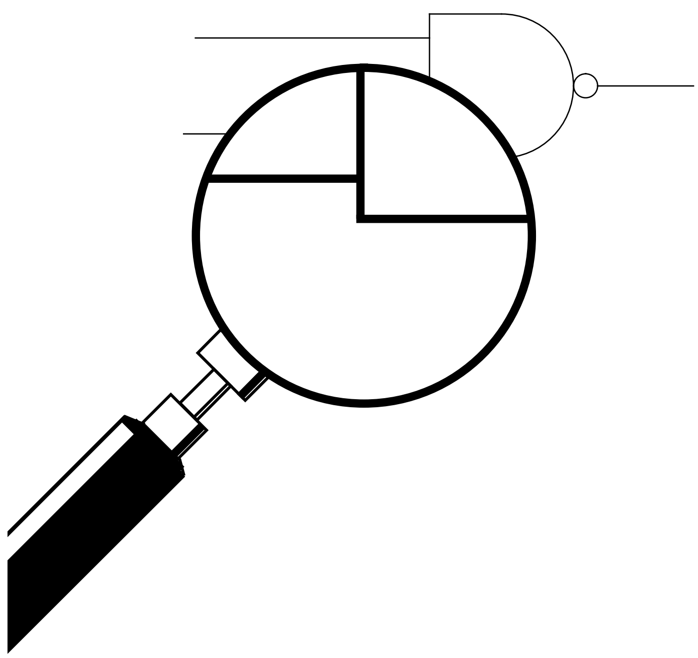
SIMIC User's Guide

Design Verification Tool



Genashor Corporation
9 Piney Woods Drive Belle Mead, NJ 08502-1109
USA
Tel: (980) 281-0164
Fax: (908) 281-9607

Genashor Corp
9 Piney Woods Drive
Belle Mead, NJ 08502
Telephone: (908) 281-0164

Table of Contents

Section 1 Fundamentals Of SIMIC Simulation

SIMIC performs tasks that are specified via a **run command** language. These commands can be entered interactively or in batch operation. SIMIC supports a large repertoire of commands to control simulation options, to modify the modeled electrical properties of circuit components, and to find and eliminate problems in the design.

SIMIC allows the designer to model a logic network by using a hierarchical or “building block” approach having as many levels as desired. Ultimately, the circuit is resolvable to an interconnection of SIMIC primitive elements, either built-in or user-defined. This hierarchical approach helps to make circuit design and debugging a more manageable task by allowing the designer to verify subcircuits independently.

A SIMIC logic network description requires only two basic statements that describe the functionality of the network; a **type** statement and a **part** statement. The **type** statement names a logic building block or “macro”, and declares its pins (inputs, bidirectionals, and outputs). The **part** statement details the composition of the macro’s components and interconnections. Each **part** statement instantiates (includes a reference to) a type which may itself be a macro.

Chapter 1.1 Basic Simulation

1.1.1 Introduction

This chapter introduces SIMIC **run commands—**instructions that are entered to control SIMIC's operation—and describes the basic steps for simulating a circuit design:

- entering SIMIC
- leaving SIMIC
- general syntax of SIMIC run commands
- the minimum set of commands needed for basic simulation
- using files that contain SIMIC run commands

Subsequent chapters of this Guide will provide greater detail on the uses and options of these commands. However, the information in this chapter is applicable to a wide range of standard situations and should provide sufficient SIMIC background to successfully begin using it.

1.1.2 Entering SIMIC

To begin a SIMIC session interactively, simply type the command:

```
simic
```

at the system's prompt.

1.1.2.1 Case Sensitivity

SIMIC is case insensitive by default—it converts all lowercase input to uppercase. In order to support case sensitive environments, SIMIC can be instructed to operate in case sensitive mode. This mode is invoked with the **-s** option on the command line:

```
simic -s
```

When this option is activated, the case of all user-defined names (i.e., signal, subcircuit, delay, and instance names) will be preserved.

1.1.3 Entering Run Commands

Upon start-up, SIMIC displays a sign-on "banner", which contains the version of SIMIC currently executing. The version number has the format:

<major-release-number> **.** *<minor-release-number>* **.** *<update-number>*

After the banner, SIMIC will display its own prompt (**>>:**) requesting command input:

```
The SIMIC Logic simulator... Version 1.00.00
Genashor Corp, Copyright 1991
>>:
```

Each time a command is entered, SIMIC will execute the command and, after all required operations have been completed, issue another prompt to indicate that it is ready for the next command.

1.1.3.1 Attention Interrupt

Most run commands specify options for subsequent simulation, and are processed virtually instantaneously, compared to human reaction time. Some operations, such as compilation and/or simulation of large circuits, and generation of voluminous reports, will require waiting for completion. If it becomes necessary to prematurely terminate these operations (e.g., a mistake has been made, too much output has been requested, unexpected results are observed early in the simulation, etc.), simply use the system Attention key (control-C on most systems). SIMIC will stop its current operation, output an ABORT message, and issue its prompt for a new command:

```
ABORT: User interrupt.
>>:
```

1.1.4 Leaving SIMIC

Once SIMIC has been invoked, a SIMIC session is ended by typing the command:

```
>>: quit
```

SIMIC will also exit if an "end of file" condition occurs during a read at the console. For example, the UNIX control-D (^D), and the VMS/MS-DOS control-Z (^Z) are equivalent to issuing a **quit** command.

1.1.5 Parts and Signals

For the purposes of SIMIC simulation, a network description consists of instantiated elements (parts) and nets (signals) that interconnect the pins of these parts. Functional attributes may be assigned to parts (boolean equations, input timing checks, etc.), and electrical attributes may be assigned to signals (rise time, decay, loading, etc.). Throughout this Guide, the terms "signal", "node", and "net" will be used interchangeably, as will "part", "component", and "element".

1.1.6 Example: The FULL-ADDER Circuit

The following example introduces basic simulation using the minimum set of SIMIC run commands. It is suggested that this circuit be referenced while reading further into Chapter 1.

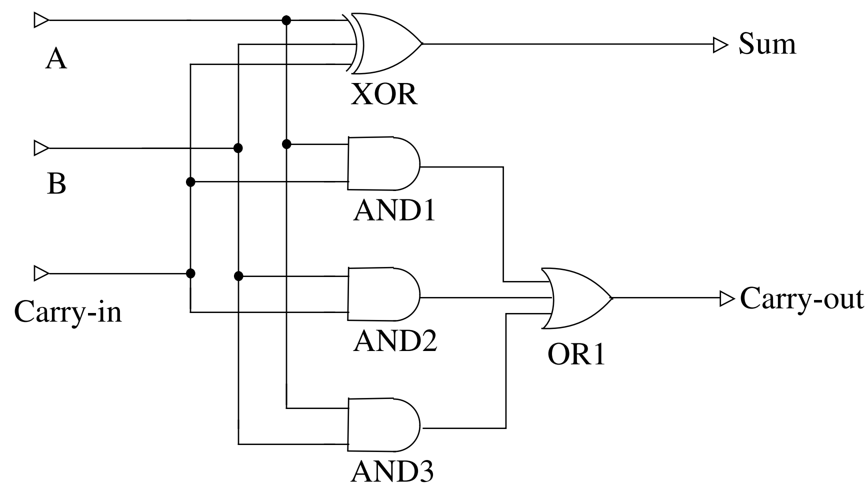


Figure 1.1-1 Full-Adder Circuit

1.1.6.1 Describing the Full-Adder

This classic arithmetic circuit, shown in Fig. 1.1-1, has three inputs and two outputs. Inputs **`a`** and **`b`** are operands, and **`carry-in`** implies connection to the **`carry-out`** of a previous stage. The three inputs are all fed to a 3-input Exclusive-OR gate, named **`xor`**, whose output, **`sum`**, is a logical-1 whenever there are an odd number of logical-1s at its inputs. The three inputs are also connected in all possible combinations to the inputs of three 2-input AND gates (**`and1`**, **`and2`**, **`and3`**) such that if two or more inputs are simultaneously logical-1, a logical-1 is propagated through the 3-input OR gate (**`or1`**) to output **`carry-out`**.

```
c= Demonstration circuit of a Full-Adder
t=full-adder i=a,b,carry-in o=sum,carry-out
   p=xor  t=exor i=a,b,carry-in   o=sum
   p=and1 t=and  i=a,carry-in
   p=and2 t=and  i=b,carry-in
   p=and3 t=and  i=a,b
   p=or1  t=or   i=and1,and2,and3 o=carry-out
```

Figure 1.1-2 SNL Description of the Full-Adder Circuit

Figure 1.1-2 illustrates a description of this circuit in SNL, SIMIC's network description language. SNL basics are covered in Chapter 1.2; however, a brief overview should suffice for a good understanding of the Full-Adder description. The fundamental concept to grasp is that SNL is a keyword-oriented language; each field of a line must be a **keyword-field** of the form ***keyword=value***. Referring to Figure 1.1-2:

1. The first line, beginning with **`C=`**, is simply a comment.
2. The second line, beginning with the **`T=`** keyword, is a **type statement** (**`T`** is an abbreviation for **`TYPE`**) that assigns the name **`full-adder`** to the circuit, defines its input (**`I=`**) pins as **`a`**, **`b`**, and **`carry-in`**, and its output (**`O=`**) pins as **`sum`** and **`carry-out`**.
3. The next five lines instantiate the five gates in the circuit, and specify the nets connected to their pins. They are called **part statements**.

   For example, the first **part statement**, line 3, places the 3-input exclusive-or gate generating **`sum`** into the circuit. The first keyword-field of this line (**`P=xor`**) assigns the name **`xor`** to this part (**`P`** is an abbreviation for **`PART`**). The second keyword-field (**`T=EXOR`**) declares the gate's type as an exclusive-or, which is one of SIMIC's built-in primitives. The third keyword-field, (**`I=a,b,carry-in`**), specifies the exclusive-or gate's three inputs. Finally, the last keyword-field (**`O=sum`**) names the gate's output signal.

Note that the output signal names of parts **`and1`**, **`and2`**, and **`and3`** are unspecified, since there is no **`O=…`** keyword-field in their associated **part statements**. If this keyword-field is omitted in a **part statement** instantiating a single-output component, SIMIC implicitly assigns the output signal the part's name. (This reduces the amount of typing required.) Thus, the names of the signals generated by the three AND gates are defaulted to **`and1`**, **`and2`**, and **`and3`**.

To further reduce the amount of typing required, SNL supports a shortcut that eliminates the need to enter keywords (e.g., **p=**, **t=**) in most **part statements**. For example, the first **part statement** could have been:

```
xor    exor   a,b,carry-in     sum
```

This method is described in Chapter 1.2.

1.1.6.2 SIMIC Simulation of the Full-Adder

Figure 1.1-3 illustrates an interactive simulation session for the Full-Adder circuit. User input is shown bold. Each user input line is terminated with the appropriate key for the current system (“Enter”, “Return”, etc.).

```
simic
The SIMIC Logic simulator... Version 1.00.00
Genashor Corp, Copyright 1991
>>: define file=fad
>>: get type=full-adder
Main Get Network : FULL-ADDER
GET completed, Circuit totals:  Parts = 5; Signals = 10
   Inputs = 3; Busses = 0; Outputs = 2
>>: define padd.3 = 000 001 010 011 100 101 110 111
>>: apply pattern=padd
>>: print list=a,b,carry-in**sum,carry-out
>>: simulate

Remark= Options: (Fault Free simulation)
Remark=   Pattern stimuli, Near Filter, Spike Propagation
Remark=   Stable Before Decay, Dynamic Delay

C=                 ABC  SC
C=                   A  UA
C=                   R  MR
C=                   R   R
C=                   Y   Y
C=                   -   -
C=                   I   O
C=                   N   U
C=                       T

        0 T     1: 000  00
        0 T     2: 001  10
        0 T     3: 010  10
        0 T     4: 011  01
        0 T     5: 100  10
        0 T     6: 101  01
        0 T     7: 110  01
        0 T     8: 111  11
>>: quit

Quit Command Issued... Leaving SIMIC
```

Figure 1.1-3 SIMIC Simulation Session For The Full-Adder

Each of the commands and the corresponding SIMIC response will be discussed in detail. To begin the SIMIC interactive session type:

```
simic
```

in response to the system prompt. Upon start-up, SIMIC prints a banner containing the version and copyright information. After this banner, SIMIC issues the interactive prompt (**>>:**), requesting user input:

```
The SIMIC Logic Simulator... Version 1.00.00
Genashor Corp, Copyright 1991
>>:
```

Step #1:

Specify default file name.

The first command entered at the prompt:

```
>>: define file=fad
```

specifies a default name, **fad**, for all files that SIMIC is requested to read or write. This default can be overridden for any file by specifying a different file name in the appropriate run command. If the default file name is not specified, SIMIC supplies the name, **noname**, as the default.

Step #2:

Specify the circuit to be simulated.

The second command:

```
>>: get type=full-adder
```

instructs SIMIC to compile the circuit, **full-adder**. Since no file was specified in this command, SIMIC uses the default file name defined in the previous command, **fad**, and the default extension for network files, **net**, as the name of the file to be read. It then searches this file for a circuit description named **full-adder**, reads the description, and compiles it. SIMIC's response to this command is:

```
Main Get Network : FULL-ADDER
GET completed, Circuit totals: Parts=5; Signals=10
   Inputs = 3; Busses = 0; Outputs = 2
```

This response reiterates the type name specified by the user and, if the compilation was successful, displays the basic circuit statistics. Note that the number of signals reported is 10, not 8 as might be expected. This is because SIMIC always includes the global signals corresponding to logical-1 (ONE) and logical-0 (ZERO). If unsuccessful, SIMIC would have enumerated the error(s) found during the compilation process, and then reported an unsuccessful **get**.

Having successfully compiled the full-adder with a **get**, the next command describes the stimuli to test the circuit. This is done with another **define** command:

Step #3a:

Specify the primary input stimuli.

```
>>: define padd.3=000 001 010 011 100 101 110 111
```

Stimulus definition will be discussed in detail in Chapter 2.3. In this example, "simulate-until-stable" **patterns** are used—one of three stimulus modes supported by SIMIC. Simply, this means that the circuit must finish responding to the current stimulus before the next stimulus change is applied. This mode of operation, also called "fundamental mode", makes verification of the circuit simpler than for other stimulus modes, where relevant to circuit operation. It also simplifies stimulus specification, since application times are not required. This run command defines a pattern sequence, named **padd**, that can be applied to 3 primary inputs. The patterns consist of a binary sequence from 0 (**000**) to 7 (**111**).

SIMIC supports hierarchical pattern definition as well as pattern definition of subsets of primary signals. Therefore, many different sequences of stimuli may be defined, each distiguished by a unique name (e.g., **padd**). Which sequence is used for simulation is selected by the following command:

Step #3b:

Apply the primary input stimuli.

```
>>: apply pattern=padd
```

This command takes the previously defined stimuli, **padd**, and applies them to the primary input signals in the order of their appearance in the Full-Adder's **type statement**. Thus, in each individual pattern of **padd**, the first bit is applied to signal **a**, the second to **b** and the third to **carry-in**. Although not used here, another keyword of the **apply** command allows specification of the correspondence between primary signals and ordering of values within the selected pattern.

The next run command instructs SIMIC to output specific simulation results at the terminal:

Step #4:

Specify what should be reported during simulation.

```
>>: print list=a,b,carry-in**sum,carry-out
```

The signals to be printed are **a**, **b**, **carry-in**, **sum**, and **carry-out**, in that order. The asterisk (*****) has special meaning in the **print** command. It is used to force a blank vertical column in the output. Therefore two blank columns will be placed between the input signals (**a**, **b**, and, **carry-in**) and the outputs (**sum**, **carry-out**). Because no other options were specified, SIMIC will, by default, output a line every time the circuit enters a stable state (all internal activity has ceased in response to an input change). Simulation output can also be written to a file with the **write** command, which will be described later in Chapter 2.4.

The next command initiates simulation:

Step #5:

Initiate simulation.

```
>>: simulate
```

SIMIC will not begin true-value simulation if no simulation output has been specified (**print**, **write**, etc.; this is not true for fault simulation). This saves the user from the embarrassment of a lengthy simulation without any output to show for it.

As shown in Figure 1.1-3, the following terminal output occurs as a result of the above run commands. It consists of (a) an options banner, displaying the currently selected simulation modes as remarks (**Remark=**), (b) a signal header, displaying the selected signals' names in vertical columns as comments (**C=**), and (c) the simulation results in a truth table format:

```
Remark= Options: (Fault Free simulation)
Remark=  Pattern stimuli, Near Filter, Spike Propagation
Remark=   Stable Before Decay, Dynamic Delay

C=                  ABC  SC
C=                    A  UA
C=                    R  MR
C=                    R   R
C=                    Y   Y
C=                    -   -
C=                    I   O
C=                    N   U
C=                        T

        0 T     1: 000  00
        0 T     2: 001  10
        0 T     3: 010  10
        0 T     4: 011  01
        0 T     5: 100  10
        0 T     6: 101  01
        0 T     7: 110  01
        0 T     8: 111  11
```

Each line of the tabular simulation output contains:

- A number indicating the total simulation time (in time-units) from the **reference time**. For this stimulus mode (patterns), the reference time is the start of each applied pattern. The displayed times are **0** in this example because delays were not assigned to element outputs; unspecified delay values default to 0. Thus, this circuit responds instantaneously to each new input pattern applied.
- The letter **T**, which serves as a delimiter between the preceding time field and the test field that follows.
- A number indicating the current test for the output line. In this case, each new pattern starts a new test.
- A colon (**:**) which serves as a delimiter between the test field and the signal values that follow.
- The signal values, which are displayed in the format specified by the **print** command.

Once simulation is complete, SIMIC issues another prompt (**>>:**), requesting further instructions. The **quit** command instructs SIMIC to exit:

Step #6:
Exit SIMIC.

```
>>: quit
```

SIMIC responds with a sign-off banner, and displays statistics for the session:

```
Quit Command Issued... Leaving SIMIC

Total SIMIC CPU-time = 0.14 sec. (00:00:00.14)
............... User : 0.12 sec. (00:00:00.12)
............. System : 0.02 sec. (00:00:00.02)
........ Page faults : 0
```

At this point, the SIMIC session is complete, and control returns to the system level.

End of Example

1.1.7 Run Command Syntax

The run command syntax is designed to be flexible, succinct, and easy to use. This section contains a brief overview of run command basics. Specific commands are described in detail throughout this Guide.

Run commands generally contain a **command verb** (or action) followed by one or more optional **keyword-fields**. Two commands, **`simulate`** and **`quit`**, may be issued without keyword options.The general form is:

<***command***> <***keyword_field***> <***keyword_field***> **. . .**

SIMIC regards tabs and space as **whitespace.**

The command verb and the keyword-fields are separated from each other (delimited) by **whitespace**, which for SIMIC is one or more tabs and/or spaces.

Run command keyword-fields qualify the action of the command verbs. A run command keyword-field is a SIMIC keyword followed by one or more characters that specify an option or value. The syntax for the keyword values depends upon the command.

Keyword-fields can take two generic forms. The first form is:

<***keyword***>=<***value_list***>

where ***<value_list>*** is a list of specific values that are associated with the keyword, delimited by whitespace and/or commas. (In some cases, where noted, other delimiting characters have significance.) For example:

`print list=a,b,c`

means “print the three signals named **`a`**, **`b`**, and **`c`**”

The second form of keyword-field is:

<***keyword***> **:**

When a keyword is followed by a colon, SIMIC will apply a default value for the keyword. In many situations, the colon may mean “all”. For example:

```
print list:
```

means “print all signals”.

Keyword values are defined by the run command in which they appear. Unless a keyword’s value is specifically modified, it will retain it’s value until the end of a SIMIC session (some return to their defaults after a new circuit description is loaded). To modify a keyword’s value that was assigned by a previous run command, the command would simply be reissued with the keyword and its new ***<value_list>***.

An action of a specified keyword may be cancelled by using the **`no`** prefix, which has the following form:

no <***command***> <***keyword_field***>

The space between the ***no*** and ***<command>*** is optional. This will *cancel* or *inhibit* the action of the command verb specified by the keyword field. An example is:

```
>>: no print list:
```

This cancels all previous **print** commands specifying signals to be reported, thereby disabling this terminal output. The **`no`** prefix is invalid for some commands, such as **`quit`**, that have no options to cancel.

1.1.7.1 Sticky Parameters

Most SIMIC run command options are “sticky”. This means that when a run command is issued, it will remain in effect until it is explicitly removed. As an example, to print the signals **`a`**, **`b`**, and **`c`**, the following command can be issued:

```
>>: print list=a,b,c
```

To cancel printing of these signals, issue the command:

```
>>: no print list=a,b,c
```

Otherwise, subsequent **`print list=`** commands will still result in printouts of **`a`**, **`b`**, **`c`**, as well as the subsequently specified signals.

Some simulation options are not sticky. They are specifically indicated as non-sticky within the remaining chapters of this Guide.

1.1.7.2 Spacing and New Lines

Each command verb must begin a new line, optionally preceded by whitespace only.

The command verb must be separated (delimited) from the first keyword by whitespace. If the command contains more than one keyword field, the keyword fields must also be separated by whitespace. Whitespace can be used freely to improve readability. Values and keywords cannot contain whitespace characters unless enclosed within single or double quotes.

1.1.7.3 Blank Lines, Line Continuation

"Line Continuation" is the one case where a blank line has relevance to SIMIC. A blank line terminates the continuation.

With one exception, blanks lines may be freely inserted in SIMIC run commands to improve readability. Lines may be of any length, but they are usually confined to 80 characters to facilitate printing and editing. In order to enter a run command spanning multiple lines, a dollar sign (**$**) is placed at the end of the line to be continued. The **$** causes the following line to be appended, after removing any leading whitespace characters. Thus if delimiting whitespace is required, it must be placed *before* the **$** character. If the **$** is followed by an equal sign (**=**), then the rest of the physical line is treated as a comment, and thus ignored. If any characters follow the **$** without an intervening **=**, then an error is reported. For example:

```
This is $
a long sen$= This text is a comment
          tence$
!
```

is read by SIMIC as:

```
This is a long sentence!
```

The exception noted above is a blank line immediately following a line being continued. Like any other continuation line, the blank line is appended to the continued line, thereby terminating it, since the blank line contains no **$** character.

1.1.7.4 Abbreviations

Abbreviations are common and useful when working with SIMIC. Most command verbs and keywords may be abbreviated to only the first two letters; more can be included. For example, the command verb **print** is abbreviated **pr**, but may also be entered as **pri** or **prin**.

However, there are exceptions to the "two-letter" rule. Three reserved words may be abbreviated to just one letter. They are: **`comment`** (**`c`**), **`zero`** (**`z`**) and **`x`** (**`x`**). Other words may require up to five letters in the minimum abbreviation to resolve ambiguity.

Throughout this Guide, whenever a new command or keyword is introduced, the minimum abbreviation will be given in parentheses, e.g., **`print`** (**`pr`**). In addition, Appendix C contains the abbreviations of all SIMIC command verbs, keywords, and reserved words.

1.1.8 RUN files

The default extension for run files is **run**.

Run commands can be placed in files and SIMIC directed to do (execute) the commands. These files are called ***run files***, and their default extension, if the full name is not specified, is **run**. For example, the commands for the Full-Adder example could have been placed in a run file called **fad**. SIMIC can be directed to read these commands by one of two methods:

1. By an interactive run command within SIMIC, **execute** **`file`** (**`ex fi`**):

   `>>: execute file=fad`

   Run files can be nested up to four levels.

   SIMIC will then proceed to do the commands in the run file, **fad**. Note that the **execute** command can also be placed into a run file, allowing "nesting" of command files. Very often it is advantageous to place the pattern definitions in a separate run file, as these can be very long. This separation allows faster modification (editing) of the other run files, since they would now be significantly smaller. If an error occurs in a command read from a run file, the error message will contain the file's name and the line number containing the error.

   More than one file may be specified in each **`execute`** command. In this case each file will be executed in the order specified. Executed run files may themselves contain **`execute`** commands, to a nesting depth of 4.

2. As a parameter on the system command line invoking SIMIC:

   `simic fad`

   This is equivalent to issuing the above **`execute`** command as the first interactive command to SIMIC.

   As in the **`execute`** command, more than one file may be specified in the command line, separated by spaces.

1.1.9 Using SIMIC in Batch (Background) Mode

To use SIMIC in batch mode, create run file(s) that contain all commands necessary for simulation, and then invoke SIMIC in the usual manner for creating background jobs. Remember to redirect console output to a file! For example, under UNIX, the command might be:

```
simic fad > fad.log &
```

and under VMS the equivalent command would be:

```
spawn/notify/out=fad.log simic fad
```

or

```
submit dosimic.com
```

where **dosimic.com** is a DCL command file containing the line:

```
simic fad
```

Under UNIX, SIMIC can also be made to read a run file by redirecting standard input (e.g., piping, '<'). *Make sure this file contains a* **`quit`** *run command*; in background mode, end-of-file may not be properly returned by the system, and SIMIC can "hang" waiting for input that will never arrive.

Chapter 1.2 Creating the Network Description

1.2.1 Types of Network Description Statements

If a Network Description File's name is specified, but not its extension, SIMIC assumes that the extension is **net**.

Before any SIMIC simulation of a circuit can be performed, a complete description of the circuit must be provided in a form that SIMIC can read. This is accomplished by creating a Network Description File using the system editor or some automated means. The default extension for Network Description Files is **net**. SIMIC's network description language is called SNL (SIMIC Network Language). SNL is easy to learn. It allows a circuit to be described with great flexibility and precision.

There are basically two types of SNL statements:

1. **Specification Statements** – These are the statements that describe circuit topology and electrical characteristics, and allow commentary. Each SNL specification statement is composed of one or more ***keyword-fields*** having the following syntax:

   <keyword>=<value_list>

SIMIC defines **whitespace** as "one or more tabs and/or spaces".

List items are usually separated by commas. Chapter 2 discusses the use of semicolons.

Keyword-fields must be separated (delimited) by **whitespace**—one or more tabs and/or spaces. Within a keyword-field, optional whitespace may be inserted before and/or after the equals sign (**=**) for readability. If ***<value_list>*** contains more than one item, the items should be separated by commas (**,**). In special cases other delimiters, such as semicolons (**;**), must be used instead of commas. As with **=**, optional whitespace may precede and/or follow these delimiters.

There are four kinds of specification statements:

- a **type statement**—declares (1) the beginning of a subcircuit (macro) definition or (2) a user-defined primitive (contains a **`type`**, but no **`part`** keyword)
- a **part statement**—instantiates a component within a macro (contains both a **`part`** and a **`type`** keyword)
- a **delay statement**—defines delay-vs.-loading characteristics (contains a **`delay`** keyword)
- an annotation statement (begins with a **`REMARK`** or **`COMMENT`** keyword).

2. **Directives** – These SNL statements control the SIMIC circuit compiler. Most directives stand alone on a line, and all must be placed at the beginning of a SNL statement, optionally preceded by whitespace. Directives that begin with an exclamation mark (`!`) control the SIMIC parser. Those that begin with a percent sign (`%`) control the SIMIC circuit compiler.

Table 1.2-1 SNL Directives

Directive	Description
!BEHAVIORAL	Declares the start of a section of statements that define behavioral model interfaces.
!DELAY	Declares the start of a section of statements that define delay-vs.-loading characteristics.
!DOCUMENTATION	Begins an annotation section (text ignored by SIMIC).
!FORMAT	Defines the section's keyword ordering.
!INCLUDE	References other files to include during compilation.
!LOGICAL	Declares the start of a section of statements that describe circuit topology.
%DECLARE	Describes collections of signals as vectors.

Every SNL file is assumed to begin with circuit topology information, thus the !**`LOGICAL`** directive is optional, unless topology is preceded by another section (**`!BEHAVIORAL`**, **`!DELAY`**, or **`!DOCUMENTATION`**).

The rest of this chapter will focus on describing topology in the **`!LOGICAL`** section, and on annotation. Chapter 2.7, which continues the description of SNL, covers circuit hierarchy and definition of electrical attributes.

1.2.2 Some Basic SNL Guidelines

1.2.2.1 Blank Lines, Line Continuation

"Line Continuation" is the one case where a blank line has relevance to the SIMIC circuit compiler.

With one exception, blanks lines may be freely inserted in SNL text to improve readability. Lines may be of any length, but they usually contain 80 characters or less to facilitate printing and editing. In order to enter a SNL statement spanning multiple lines, a dollar sign (**$**) is placed at the end of the line to be continued. The **$** causes the following line to be appended, after removing any leading whitespace. Thus, if delimiting whitespace is required, it must appear *before* the **$** character in the line being continued. If the **$** is followed by an **=**, then the rest of the physical line is treated as a comment, and thus ignored. If any characters follow the **$** without an intervening **=** then an error is reported. For example:

```
This is $
a long sen$= This text is a comment
          tence$
!
```

is read by SIMIC as:

```
This is a long sentence!
```

The exception noted above is a blank line immediately following a line being continued. Like any other continuation line, the blank line is appended to the continued line, thereby terminating it, since the blank line contains no **$** character.

1.2.2.2 User-defined names

Any names that are created must begin with either an underscore (**_**), question mark (**?**) or an alphanumeric character (0-9, a-z, A-Z). The remainder of the name can contain more alphanumeric characters, underscores, and question marks, as well as hyphens, percent signs, exclamation marks, and periods. However, if any other characters are to be used, such as a pound sign (**#**), or a backslash (****), then the name (or the portion containing the special character) must be enclosed in single (apostrophe) or double quotes.

In case-insensitive mode, double-quoting a name preserves the character's case (upper or lower). Otherwise all lowercase letters (a-z) are converted to upper case.

In case-insensitive mode (when SIMIC is not invoked with **-s** option—see *Entering SIMIC* in Chapter 1.1), SIMIC converts all user-defined names to uppercase unless directed otherwise. Double quotes serve the dual role of providing this direction; the case of text enclosed within double quotes is preserved. Below are some examples:

original:	read as:
abc	ABC
"abc"	abc
'abc'	ABC
ab"c"	ABc
'ab'"c"	ABc
"ab'c'"	ab'c'
'ab"c'	AB"C

There are three reserved signal names in SIMIC which have special meaning, and should not be used as the name of any primary or internal signals.

The three names are: **`ONE`**, **`UNUSED`** and **`ZERO`**, corresponding to logical-1, an unused component pin, and logical-0, respectively. The proper use for these reserved signal names is illustrated in examples throughout this Guide.

It is important to note that the reserved signal names will match the correct spelling, even if the cases don't match. Therefore: "one", "One", "ONE" etc. are all equivalent. Similarly, the names of SIMIC primitives are case insensitive; for example, "and", "And", "AND" etc. are synonymous.

1.2.3 Describing Circuit Topology

1.2.3.1 !LOGICAL

A **`!LOGICAL`** directive is implicitly assumed at the beginning of each Network Description File (default file extension **net**). Therefore, if the NET file begins with a topological description section, it is not necessary to place the **`!LOGICAL`** directive at the beginning of the file.

The **`!LOGICAL`** section describes the circuit structurally, that is, as an interconnection of SIMIC primitives (built-in or user-defined) and **macros**, which are subcircuits containing primitives and/or other macros.

The circuit is described using a hierarchy of **`type`** and **`part`** specification statements. Each **`!LOGICAL`** section contains the definition of one or more **type**s. Each such definition is called a **type block**.

1.2.3.2 Basic Type Block Structure

A **type block** is composed of a single **type statement** followed by one or more **part statements**. The one exception to this the BOOLEAN **type block**, which has no associated **part statements**.

type statement description.

This statement begins the definition of a subcircuit (macro).

A **type statement** declares the beginning of a macro description, assigns the macro a name, defines its external pins—inputs, outputs, and busses (bidirectionals), and optionally specifies electrical attributes (e.g., delay, pin loading). The simplest form of **type statement** is:

```
TYPE=<type_name> I=<input_list> $
O=<output_list>
```

where:

- ***<type_name>*** is the user-defined name of the macro being defined.
- <***input_list***> is the list of user-defined names for input pins, one name per pin, separated by commas (,).
- <***output_list***> is the list of user-defined names for output pins, one name per pin, separated by commas (,).

part statement description.

This statement places a part within the **type block** and connects its pins.

A **part statement** instantiates (places) a component within the **type block**. It assigns an instance name to the component, which must be unique within this **type block**. It specifies the type of component being instantiated (e.g., AND, NOR, or the type name of a macro defined elsewhere). It also specifies the interconnections with the other parts within the **type block**, or with the pins of the **type block** itself. Finally, it may also describe electrical characteristics of this component, such as output delay, pin loading, etc.

When considering **part statements**, it is vital to make a clear distinction between the **type being defined** (the ***<type_name>*** in the **type statement**) and the **type being instantiated** (the value of the **`TYPE=`** keyword-field of the **part statement**).

Functionally (or *semantically*), **type statements** and **part statements** have totally different functions.

Structurally, (or *syntactically*), the main difference between the two is that a **type statement** must contain a **`TYPE=`** keyword-field, but not a **`PART=`** keyword-field, while a **part statement** must contain both a **`TYPE=`** keyword-field and a **`PART=`** keyword-field.

The simplest form of the **part statement** is:

```
PART=<part_name> TYPE=<referenced_type> $
I=<input_list> O=<output_list>
```

where:

- ***<part_name>*** is the user-defined name of this instance of the ***<referenced_type>***. This name must be unique within the **type block**.
- ***<referenced_type>*** is the name of the SIMIC primitive (built-in or user-defined) or macro to be instantiated as this PART. If the circuit compiler cannot recognize the referenced type as a primitive or as a macro defined elsewhere, then an error is issued.
- ***<input_list>*** is the list of user-defined names for input nets—one name per net, separated by commas (,).
- ***<output_list>*** is the list of user-defined names for output nets,—one name per net, separated by commas (,).

Each user-defined net name specified in ***<input_list>*** or ***<output_list>*** is associated in a one-to-one correspondence with ***<referenced_type>***'s defined pin order. For example, DL is a built-in SIMIC D-latch primitive. Its input pins are defined as **`NR`**, **`NS`**, **`C`**, **`D`**, in that order (active-low reset, active-low set, clock, and data, respectively), and its single output pin is defined as **`Q`** (see Appendix A). The **part statement**:

```
PART=q TYPE=dl I=reset,set,clock,data O=l_out
```

instantiates a DL whose part name is **`q`**, which must be a unique part name within the containing **type block**. The DL's **`NR`** pin is connected to a net named **`reset`**, its **`NS`** pin to **`set`**, its **`C`** pin to **`clock`**, and its **`D`** pin to **`data`**. Its single output pin, **`Q`**, is connected to signal **`l_out`**.

1.2.3.3 Implicit Names And Connections

A signal is implicitly connected to an identically-named pin of the **type being defined**.

No associations are made for identical *user-assigned* signal names, part names, and pin names of **instantiated types**.

Connectivity between the internal nets and pins of the **type being defined** is established by the commonality of their names. In other words, pins declared in the **type statement** and nets with the same name are implicitly connected together within the same **type block**.

In contrast, no implicit association is made between user-assigned names and pin names of *instantiated* **types** that happen to be identical. Thus, in the above example, even though the assigned part name, **q**, is identical to the DL primitive's output pin name, no ambiguity is introduced, nor connectivity implied, by assigning this particular part name.

To reduce the amount of typing required, SIMIC supports an implicit naming convention for certain output signals. If a **part statement** instantiates a type that has <u>only one output</u>, then the **O=** keyword-field may be omitted. In this case, the output net will have the <u>same name</u> as its part. For example:

```
PART=q TYPE=dl I=reset,set,clock,data
```

is equivalent to:

```
PART=q TYPE=dl I=reset,set,clock,data O=q
```

For a part having only one output, the output name defaults to the part's name if the **O=** keyword value is omitted.

Any input pin of a part may be connected to logical-0 or logical-1 by using the reserved words **ZERO** and **ONE**, respectively. For example:

```
PART=q TYPE=dl I=reset,one,clock,data
```

connects the DL's **NS** pin to **ONE**, thereby disabling this input.

1.2.3.4 Advanced Topic Digression: By-Name Pin Connections

This topic is placed here for completeness and is not prerequisite for future reading.

An <u>alternative form</u> for specifying a part's pin connections utilizes connection by-pin-name (named association) rather than by-order (positional association). This is important for some schematic capture programs that do not have an ordering attribute for pins. However, the by-order convention is the more compact form, simplifying manual entry.

The simplest form for the by-pin-name connection syntax is:

<***net***> (<***pin***>)

where:

- ***<net>*** is the net name to connect to the designated ***<pin>.***
- ***<pin>*** is the pin name of the primitive or macro.

For example:

```
PART=q TYPE=dl $
I=reset(nr),set(ns),clock(c),data(d)
```

connects the net named **reset** to the pin **nr**, **set** to **ns**, etc.

There must not be a white-space character before the left parenthesis in either form.

The full form for by-pin-name connections is:

`'<'` <***net_list***> `(`<***pin_list***>`)` `'>'`

where:

- ***<net_list>*** is a list of net names separated by commas to connect to the ***<pin_list>***
- ***<pin_list>*** is a list of pin names of the instantiated primitive or macro.
- **‘<’** and **‘>’** are literals, rather that syntactic descriptors, indicating that left angle bracket and right angle bracket, respectively, must be placed at these positions.

The enclosing left (<) and right (>) angle brackets are required only if ***net_list*** contains commas; otherwise they are optional. For example, the following **part statement**s are identical to the one in the previous example:

```
PART=q TYPE=dl $
  I=<data,clock(d,c)>,<reset,set(nr,ns)>
PART=q TYPE=dl $
  I=<data,reset,set,clock(d,nr,ns,c)>
```

1.2.3.5 SIMIC Primitives

A ***primitive*** is an element that SIMIC can model directly, without requiring a structural description containing simpler functions. SIMIC supports a number of built-in primitives that can be used as a basis for describing circuits:

- simple combinational gates – inverter, and, nand, or, nor, exclusive-or, exclusive-nor
- combinational functions – and-and-nor, or-or-nand, multiplexer
- latches – nand-latch, nor-latch, D-latch
- edge-triggered flip-flops – D, JK, T
- tristating drivers
- programmable element and memories – PLA, ROM, RAM
- bidirectional and switches – ideal and resistive
- backannotation – wiring capacitance and path-delay.

A full description of each built-in SIMIC primitive can be found in Appendix A of this Guide.

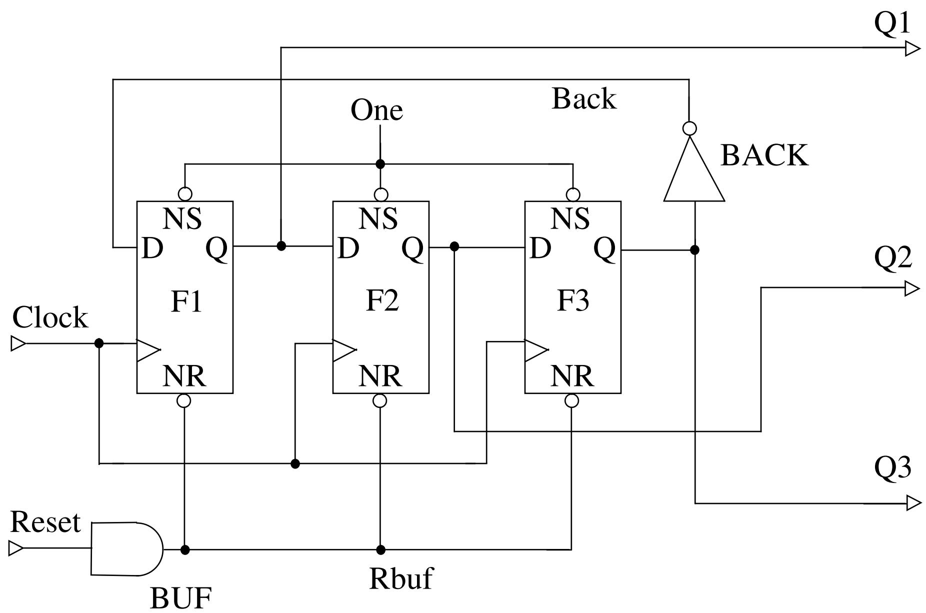
Primitives may also be user-defined. SIMIC supports defining new primitives by Boolean equations in **type** or **part statements**.

To use a primitive in a circuit description, enter the primitive’s type name as the value of the **`TYPE=`** keyword-field in the **part statement** (e.g.

TYPE=and). Built-in primitives that represent simple combinational gate functions (AND, OR, NAND, NOR, EXOR, EXNOR) may have from 1 to 32767 inputs, which can be specified in any order. However, for primitives whose inputs (or outputs) have dissimilar functions (such as flip-flops), or for user-defined primitives, it is very important to specify net names for the **I=** keyword in the exact order that SIMIC expects, when using by-order specification. The expected pin order for each built-in primitive is given in the associated Type Statement in Appendix A.

1.2.3.6 Example

Figure 1.2-1 illustrates a 3-stage Johnson Counter, and a SNL description of this circuit. It contains three (3) rising-edge triggered D flip-flops, an inverter, and a buffer on the reset line. The **part statement**s within the SNL description are indented for readability.




```
TYPE=johnson_counter I=clock,reset O=q1,q2,q3
   PART=buf TYPE=and I=reset O=rbuf
   PART=f1 TYPE=dcf I=rbuf,one,clock,back O=q1
   PART=f2 TYPE=dcf I=rbuf,one,clock,q1 O=q2
   PART=f3 TYPE=dcf I=rbuf,one,clock,q2 O=q3
   PART=back TYPE=inv I=q3 O=back
```

Figure 1.2-1 Three Stage Johnson Counter And An Equivalent SNL Description

type statement example.

In describing the circuit to SIMIC, begin with the **type statement**, which declares the start of a macro definition. Again, use the **TYPE=** keyword-field to name the type, and the **I=** and **O=** keyword-fields to name the inputs and outputs, respectively.

```
TYPE=johnson_counter I=clock,reset O=q1,q2,q3
```

This **type statement** declares that the current **type block**, consisting of this statement and the **part statements** which follow it, is a description of a macro called **`johnson_counter`**, having five pins—two inputs (**`clock`** and **`reset`**) and three outputs (**`q1`**, **`q2`**, and **`q3`**).

A **type statement** must be followed by one or more **part statement**s (with the exception of the BOOLEAN type statement), each of which: (a) instantiates a primitive or macro, (b) gives this instance a unique part name, and (c) specifies the signals connected to its pins. The **part statements** for the **`JOHNSON_COUNTER`** are:

part statement
example

```
PART=buf TYPE=and I=reset O=rbuf
PART=f1 TYPE=dcf I=rbuf,one,clock,back O=q1
PART=f2 TYPE=dcf I=rbuf,one,clock,q1 O=q2
PART=f3 TYPE=dcf I=rbuf,one,clock,q2 O=q3
PART=back TYPE=inv I=q3 O=back
```

`johnson_counter` has five parts, all of which are built-in primitives. Examining each of the five **part statement**s:

- The first **part statement** creates a part called **`buf`**, which is an instance of the AND primitive. Only one input is listed, specifying a connection to **`reset`**. Because **`reset`** is one of **`johnson_-counter`**'s external inputs, SIMIC "knows" that the input of **`buf`** is to be connected to the **`reset`** input pin of **`johnson_counter`**. Part **`buf`** has one output signal named **`rbuf`**. Since this is not one of **`johnson_counter`**'s external pins, SIMIC "knows" that **`rbuf`** is an internal node.
- The next three parts (**`f1`**, **`f2`**, **`f3`**) are instances of the DCF built-in primitive. The DCF is a positive-edge-triggered D flip-flop (see Appendix A), and its input pins (**`NR`**, **`NS`**,**`C`**, **`D`**—in that order) each have a unique function. Input signal names specified in the **`I=`** keyword-field are associated with the DCF primitive's input pins in exactly this order. Thus, the **`NR`** (active-low asynchronous reset) pin of each DCF is connected to **`rbuf`**, which is the output of the part **`buf`**, described in the previous paragraph. The **`NS`** (active-low asynchronous set) pins are all connected to **`ONE`**; therefore, these pins will be held at a logical-1 throughout the simulation. This effectively disables the **`NS`** port of each DCF. All the **`C`** (clock) pins are connected to **`clock`**, one of **`johnson_counter`**'s external inputs. Part **`f1`** has its **`d`** (data) input connected to the net named **`back`**, while the **`d`** inputs of parts **`f2`** and **`f3`** are **`q1`** and **`q2`**, respectively. The outputs of the three DCFs, **`q1`**, **`q2`**, and **`q3`**, are also connected to the output pins of **`johnson_counter`**.

- The last **part statement** creates an instance of the built-in primitive INV (an inverter) called **`back`**. Its single input is connected to the output of part **`f3`**, which, again, is the external output pin, **`q3`**, of **`johnson_counter`**. The output of part **`back`**, also called **`back`**, is connected to the **`d`** input of part **`f1`**. Note that, since the part name and output net name are identical, this **`O=`** keyword-field could have been omitted. Note also that this net (**`back`**) was previously referenced in the part statement for **`f1`**.

End of Example

Cautionary Note: it ***is*** legal to assign the same signal name to the outputs of different parts—this causes the outputs to be wire-tied.

Again, notice that in the Johnson counter example, each element and signal are given unique names. Each part within a **type block** has a unique name, as does each **type block** pin. Moreover, in a circuit with multiple **type block**s (this Johnson Counter only has one), each type name, specified in a **type statement**, must be unique.

SIMIC supports many options for describing the attributes of external pins and internal nets for any circuit (or subcircuit). A full list of the SNL keywords that SIMIC will recognize appears in Appendix B. It includes such attributes as **`OUTPUT-DRIVE`**, **`INPUT-LOAD`** and a host of other parameters useful in modeling a circuit (also see Chapter 2.7). If unspecified, SIMIC supplies default values for all electrical attributes.

1.2.3.7 Keyword-field Ordering

One advantage of a keyword-oriented description language is that the order of the fields within each statement is not important. As an example, the following is a functionally identical restatement of the **`johnson_counter`** **type block** which illustrates SNL's flexibility with regard to **part statement** and keyword-field ordering.

keyword-field order flexibility example.

```
TYPE=johnson_counter O=q1,q2,q3 I=clock,reset
 I=q3 O=back PART=back TYPE=inv
 TYPE=and I=reset O=rbuf PART=buf
 I=rbuf,one,clock,q1 PART=f2 TYPE=dcf O=q2
 I=rbuf,one,clock,q2 TYPE=dcf PART=f2 O=q3
 TYPE=dcf I=rbuf,one,clock,back PART=f1 O=q1
```

It is suggested, however, that a consistent ordering be used to improve readability.

1.2.3.8 Abbreviations

In order to reduce typing, SIMIC recognizes abbreviations for almost all of the SNL keywords. Some abbreviations were already used to describe the **johnson_counter**. The following is a list of the keywords used in their full and abbreviated forms:

Full:	Abbr.:	Notes:
INPUT-PINS	IPINS,I	Used in **type statements**
INPUT-NETS	INETS,I	Used in **part statements**
OUTPUT-PINS	OPINS,O	Used in **type statements**
OUTPUT-NETS	ONETS,O	Used in **part statements**
PART	P	
TYPE	T	

When a new keyword is introduced in the remainder of this Guide, its valid abbreviations will be given. A complete list of SNL keywords and their abbreviations can be found in Appendix B of this Guide.

1.2.3.9 !FORMAT (!FOR !F)

To further reduce the amount of typing required, as well as entry time and file sizes, SNL provides a method for eliminating even the abbreviated keywords. The **!FORMAT** directive specifies the order of keyword-fields in subsequent specification statements. Once this order has been established, only the values need to be entered rather than the entire keyword-fields.

Figure 1.2-2 below illustrates the SNL description of the **johnson_-counter** using the **!FORMAT** statement to order keyword-fields within **part statements**:

!FORMAT example.

```
!FORMAT    P=         T=         I=         O=
T=johnson_counter I=clock,reset o=q1,q2,q3
 buf       and        reset                 rbuf
 f1        dcf        rbuf,one,clock,back   q1
 f2        dcf        rbuf,one,clock,q1     q2
 f3        dcf        rbuf,one,clock,q2     q3
 back      inv        q3
```

Figure 1.2-2 Johnson Counter Description Using !FORMAT

In this SNL description, the **!FORMAT** statement declares that the first field in each subsequent **part statement** will be the part name (**P=**), the second field will be the referenced type (**T=**), the third will specify the part's input(s) (**I=**), and the fourth field will specify the part's output(s) (**O=**). When, for example, the **part statement** instantiating the DCF named **f1** is read, the first keyword of the **!FORMAT** statement, **P=**, associates with the first item in the line, **f1**, to form the complete keyword-field **P=f1**. Next, the second format keyword, **T=**, combines with the line's second item, DCF.

Similarly, the third format item, **I=**, combines with the line's third item, which is the list **rbuf,one,clock,back**, and **O=** combines with **q1**.

The following are guidelines for using the **!FORMAT** directive:

1. A **!FORMAT** statement may be defined uniquely for the **!DELAY** and **!LOGICAL** sections.
2. A **!FORMAT** statement remains in effect for a section until the next **!FORMAT** statement in that section. This is true even when multiple Network Description Files are used. **!FORMAT** statements remain in effect from file to file.
3. To cancel the **!FORMAT** statement, issue a **!FORMAT** statement without any keywords following it.
4. If a statement does not require a value for one of the keywords specified in the current **!FORMAT** statement, enter a hyphen (**–**) as a "placeholder" for that keyword's value to skip by the keyword.
5. If no values are required for keywords at the end of the **!FORMAT** list, hyphens are not required, since no keywords are skipped.
6. In contrast, if a statement requires a keyword-field that is not specified by the current **!FORMAT** statement, or the keywords need to be specified in a different order, the entire keyword-field must be typed in. This will suspend the formatting for the rest of the statement. Therefore, all subsequent keyword-fields must be explicitly entered (of course abbreviations are acceptable). Clearly, the most efficient way to add a keyword-field is to append it to the end of the line.

The following example demonstrates the various usages of formatted **part statements**. In this example, all the **part statements** are equivalent:

```
!format   part=     type=     i=        o=
          -         and       a,b part=c
          c         and       a,b
          c         and       i=a,b     o=c
          part=c type=and i=a,b o=c
```

The first **part statement** illustrates guidelines (4) and (6). The hyphen skips over the **!FORMAT** statement's **PART=** keyword, so its **and** entry is associated with the **!FORMAT** statement's **TYPE=** keyword. The entry after the input signal field, (**a,b**) would normally associate with the **!FORMAT** statement's **O=** field, but the **PART=c** entry, containing a keyword, overrides formatting.

The second **part statement** illustrates guideline (5). Here, the **O=** entry was omitted (since the output signal's name will default to the identical part name), but no placeholder hyphen was necessary because no other fields follow.

The third **part statement** illustrates guideline (6). Having used a keyword to specify the inputs (**`I=A,B`**), formatting is suspended, and a complete keyword field (**`O=c`**) is required to specify the output signal.

1.2.3.10 Signal Arrays

Signals can be grouped into arrays of up to 2 dimensions. The terms “array” and “vector” will be used synonymously throughout the remainder of the Guide.

Arrays are declared with the **`%DECLARE`** statement. The scope of a **`%DECLARE`** statement is local to the type in which it is contained. In addition to explicitly declaring array bounds, this statement also associates a default radix with each array for subsequent display during simulation. The format for the **`%DECLARE`** statement is:

`%DECLARE` *<format>*=*<array list>*

where ***<format>*** is one of the following:

- **`LEVEL (LEV)`** -- standard level format.
- **`OCTAL (OCT)`** -- octal format.
- **`HEXADECIMAL (HEX)`** -- Hexadecimal format.
- **`INTEGER1 (INT1)`** -- One’s complement format.
- **`INTEGER2 (INT)`** -- Two’s complement format.
- **`POSINTEGER (POSINT)`** -- Positive integer format.

and ***<array list>*** is a list of arrays (separated by commas), each array in the format:

<root name> [*<range #1>*]

for single dimensional arrays, and:

<root name> [*<range #2>*] [*<range #1>*]

for two-dimensional arrays. Here, ***<root name>*** is the user-assigned name for the array, and ***<range>*** specifies the extents for the indicated array dimension in the format:

<start> **:** *<end>*

where ***<start>*** and ***<end>*** are integers describing the starting and ending limits of the array.

If declared, an array can be used in the type statement or part statement as follows:

1. If only the ***<root name>*** is entered, then the entire array will be substituted for ***<root name>***, in the declared order.
2. For any other array selection, all dimension ranges must be specified.
3. If the ***<start>*** and ***<end>*** values are the same, then the colon (:) and ***<end>*** values may be omitted.

4. Only signals may use array notation (i.e not parts or types).

For example, the outputs of the flip flops in the **johnson_counter** can be declared as an octal array in the following manner:

```
!FORMAT   P=        T=        I=        O=
T=johnson_counter I=clock,reset o=q
%declare octal=q[1:3]
buf       and       reset               rbuf
f1        dcf       rbuf,one,clock,back q[1]
f2        dcf       rbuf,one,clock,q[1] q[2]
f3        dcf       rbuf,one,clock,q[2] q[3]
back      inv       q[3]
```

The **%declare** statement specifies that **q** consists of three signals: **q[1]**, **q[2]**, and **q[3]**, with a default display format of octal. The output in the type statement is **q**, which represents all signals of the array in their declared order. The following is an equivalent type statement:

```
T=johnson_counter I=clock,reset o=q[1],q[2],q[3]
```

Each **dcf** part statement specifies the individual array component that is the corresponding flip-flop's output.

For further information on the display radix and its use, see the Section *Controlling Signal Value Representation* in Chapter 2.4.

1.2.4 Annotation

It is highly recommended that comments and documentation be added to the Network Description (NET) file. This will not only make it easier to keep track of decisions made in describing a circuit, but it is immeasurably helpful to anyone else who may need to understand this file.

SNL provides three ways to annotate a file.

1.2.4.1 The REMARK (REM, R) keyword

Case is preserved in a **`REMARK`**.

`REMARK` is a SNL keyword that allows a message to be sent to the console while the circuit is being compiled. The **`REMARK`** should be placed within the **type block** requiring the annotation (i.e. below the **type statement**). Remarks are displayed at the console during circuit compilation. SIMIC ignores remarks that are not in a **`!LOGICAL`** section. Textual case within a **`REMARK`** statement is preserved.

The dollar sign (**`$`**) continuation character is not supported for remarks. In order to continue a remark beyond one line, a **`REMARK`** keyword must start the remark on the following line. The following example illustrates remark continuation.

```
TYPE=my8590 I=i1,i2,i3 O=o1,o2
P=p1 T=8590 I=i1,i2,i3 O=o1,o2
REMARK= Caution: the 8590 cell is obsolete,
REMARK= please use the 8690 cell in all
REMARK= new designs.
```

During circuit compilation, only remarks in **type blocks** actually used in the circuit will be displayed, and then only once, no matter how many times the **type** is instantiated.

1.2.4.2 The COMMENT (COM, C) Keyword

Frequently, brief commentary can be very helpful. If the commentary will only occupy a few lines, the **COMMENT** keyword is appropriate. Comments are ignored by SIMIC.

The dollar sign (**$**) continuation character is not supported for comments. All text from the **COMMENT=** keyword to the end of the physical line is ignored. The **COMMENT** keyword-field can appear anywhere in the Network Description File; either at the end of **part**, **type**, or **delay** statements, or in lines between these statements.

Unlike remarks, comments are not displayed during circuit compilation. The following demonstrates the use of comments:

```
C= ************************************
C= * An Example of some ways to include *
C= * commentary.                        *
C= ************************************

C= Comments may be put anywhere in a NET file,
C= as long as you remember that '$' is ignored
C= in comments, and that all comments begin
C= with the COMMENT keyword.

!LOGICAL C= The network description section
TYPE=buf I=in O=out C= Another comment!
```

1.2.4.3 The $= Comment

Since dollar signs (**$**) are ignored in comments, a special construct is provided in order to put comments inside continued statements. If an equals sign (**=**) is appended to the **$** continuation character, then the remainder of the line will be treated as a comment and the statement will properly continue on the next physical line. If the **=** is omitted, then a warning will be issued if anything but whitespace is found before the physical end of line. Following is a simple example of continuation comments:

```
TYPE=johnson_counter $= type block name
I=clock,reset $= Inputs for this type
O=q1,q2,q3 C= Outputs for this type
```

Note that if the last comment used the **$=** comment instead of the **C=** comment, then a blank next line would have been required to properly terminate the statement.

1.2.4.4 The !DOCUMENTATION (!DOC) Directive

The SNL **!DOCUMENTATION** directive provides the opportunity to make extensive comments in the Network Description (NET) file without having to begin each line with the **COMMENT=** keyword. SIMIC will ignore everything from the **!DOCUMENTATION** statement to the next section header (**!LOGICAL**, **!DELAY**, etc.) or the end of the file. For example:

```
!DOCUMENTATION
Many designers find it a tedious and
distracting task to document their work,
despite the obvious benefits it provides to
themselves and to other designers.

!LOGICAL C= Start of Network Description
!FORMAT   P=        T=        I=        O=
T=johnson_counter I=clock,reset O=q1,q2,q3
buf       and       reset               rbuf
f1        dcf       rbuf,one,clock,back q1
f2        dcf       rbuf,one,clock,q1   q2
f3        dcf       rbuf,one,clock,q2   q3
back      inv       q3

!DOCUMENTATION
SIMIC offers a variety of Annotation and
Documentation options.
```

Since a directive declaring a new section (**!LOGICAL** or **!DELAY**) implicitly terminates the **!DOCUMENTATION** section, the first character in each line of annotation should never be an exclamation mark.

Section 2   SIMIC Simulation

This chapter introduces you to all aspects of SIMIC good-logic (fault-free) simulation. Each section provides detailed information on a specific topic, and is as self-contained as possible. Thus, you can read these sections in any order, since no section is a prerequisite for another. However, the material in Chapter 1 should be well understood, since the current chapter builds on this basis.

Chapter 2.1 File Naming Conventions

SIMIC reads and creates a variety of textual and binary files. This section describes SIMIC file naming conventions, and illustrates how to specify operating system dependent file names to SIMIC.

2.1.1 File Name Format

2.1.1.1 File Names and Extensions

SIMIC maintains a file naming convention that is oriented toward classifying files by project and function. Every file's name has two components; a **name** and an **extension**.

SIMIC supports an implicit (default) convention that associates the file's name with the project (typically, the name of the circuit being simulated) and its extension with file's contents.

Alternatively, the user can specify file names explicitly in each SIMIC run command that reads or writes a file, to assign file names in another manner. In this case, the format for completely specifying a file's name to SIMIC is:

<***name***> **.** <***extension***>

where ***<name>*** and ***<extension>*** follow the rules for valid file names in the current operating environment.

Most operating systems support the use of dot (.) in file names. For these systems, SIMIC reads and writes files whose names have the above format, whether file names are implicitly or explicitly specified. For those environments that use another format (such as IBM/CMS, which uses a space to delimit ***<name>*** and ***<extension>***), SIMIC still expects the above format to maintain consistency. Internally, SIMIC automatically converts file names to the proper system representation.

2.1.1.2 File Names And System Compatibility

SIMIC file name and extension specifications may be constructed from:

a. any sequence of characters beginning with an underscore (_), question mark (?) or an alphanumeric character (0-9, a-z, A-Z) and optionally containing more characters that are alphanumeric characters, underscores, question marks, hyphens, percent signs, exclamation marks, and periods, or

b any sequence of characters enclosed in single or double quotes.

However, the main consideration in selecting file names is compatibility with the operating system's file naming conventions; in general, there should be considerable overlap of valid system file names and valid SIMIC file names. Some valid system file names, though, may not be compatible with SIMIC; in order to specify a name that does not conform to the above rules, the name should be enclosed in either single or double quotes.

When SIMIC is operating in case-insensitive mode, choice of which quotes to use, or even whether to enclose valid SIMIC names in double quotes, depends on the operating system. Since some operating systems are case sensitive, SIMIC uses the following convention when operating in this mode:

- The case of all text enclosed in double quotes (") is preserved.
- If a name contains no double quotes, it is converted to lowercase.
- If substrings of a name are enclosed in double quotes, and others aren't, the substrings outside the quotes are converted to uppercase.

Of course, when SIMIC is operating in case-sensitive mode, (**-s** command line option) no case conversion occurs, so user-defined names must be directly system compatible.

2.1.2 Implicit File Names

If SIMIC needs to open a file whose name is not explicitly specified, it creates the name from a common user-supplied name and an extension that is dedicated to the run command being executed. This common name, called the **default file name**, is specified with the **`FILE`** (**`FI`**) keyword option of the **`DEFINE`** (**`DE`**) run command:

```
DEFINE FILE=<name>
```

Note that *no* extension is specified here, since this will depend on the run command.

Every SIMIC run command that reads or writes a file has an associated keyword for specifying the file's name (e.g., **`FILE`**, **`LFILE`**, **`SFILE`**). If this keyword is suffixed with a colon (**`:`**), then SIMIC will utilize the file's implicit name.

For example, as a result of the following run commands:

```
DEFINE FILE=proj1
GET LFILE:
WRITE FILE:
```

the listing file, **`proj1.lst,`** will be created by the **`GET`** command, and the simulation output file, **`proj1.wrt`**, will be created by the **`WRITE`** command.

The default file name can be changed at any time by entering a new **DEFINE FILE** command. For example, if the following run commands are issued after those shown above:

```
DEFINE FILE=proj2
WRITE FILE:
WARN FILE:
```

then subsequent simulation output will go to **proj2.wrt**, instead of **proj1.wrt**, and simulation warning messages will be written to **proj2.wrn**.

Note: The default file name itself has a default; if no **DEFINE FILE** command has been issued, the default file name is **noname**.

2.1.3 Explicit File Names

File names are explicitly specified by using the equals (**=**) form of run command keyword-fields instead of the colon (**:**) form. For example,

```
DEFINE FILE=proj1
WRITE FILE=proj2
WARN FILE:
```

explicitly specifies the simulation output file to be **proj2.wrt**, and implicitly specifies the warning message file to be **proj1.wrn**.

Note that, in this example, the simulation output file's default extension, **wrt**, was implicitly used even though the file's name was explicitly specified. If desired, the file extension can also be specified explicitly. For example,

```
WRITE FILE=proj2.writefile
```

specifies that the simulation output file is **proj2.writefile**. Here, the full file name, ***<name>.<extension>***, has been specified.

2.1.4 Spanning Directories

If it is necessary to read or write files in a different directory, their names should be enclosed in either single or double quotes (depending on the operating system—see the Section *File Names and System Compatibility* above), since the characters necessary to construct path or directory names are generally not valid characters for constructing SIMIC file names. File names containing directory names are system dependent. Examples:

```
WARN FILE="/home/projects/proj1.wrn"
WARN FILE='USER1:[MITCH.PROJECTS]PROJ1.WRN'
```

Both examples specify a file named **proj1.wrn** in a subdirectory named

`projects`; the first example would be entered under the UNIX environment, the second under VMS. Note that the file extension must be specified in this format.

Chapter 2.2   Circuit Compilation

2.2.1   Introduction

Circuit compilation is the process translating a circuit's textual SNL description into a binary representation that SIMIC can use during simulation. In the process, many topological and electrical checks are performed, and a number of circuit modifications and optimizations are made to improve simulation throughput and accuracy:

1. Optimization and consolidation:
    a. Simple combinational gates and switches that are physically paralleled are merged into a single device, and the electrical characteristics are modified accordingly.
    b. Bidirectional switches are converted to unidirectional switches, where appropriate. This can dramatically improve the performance of switch level networks.
    c. Only the TYPEs that are actually used in the design are compiled. Besides improving compilation performance, this has the added benefit of reducing the amount of memory required for the final simulation structures.
2. Delays are computed from delay curves and loading factors. Auxiliary procedures and/or programs are not required to produce accurate delays.
3. Physical size metrics (number of transistors, total width of cells, and number of bond pads) are computed and reported. This information can be useful for size estimates by designers and/or place and route programs.
4. Many topological checks are performed, a few being:
    a. Incorrect number of input, output or bus connections.
    b. Switch connection between power rails.
    c. Both ports of switches connected to same node.
    d. Switch permanently turned off (disabled by control value).
    e. Nodes that don't have a driving element attached.
    f. Nodes that don't have any elements attached.
    g. Nodes with only driving elements attached.
    h. Wire-tied nodes with non-tristating driving element attached.

    Some situations, such as incorrect number of pin connections, are fatal, and prevent the completion of circuit compilation. Others, such as wire-tied non-tristating drivers, cause SIMIC to generate warning messages that may indicate potential problems with the description. These messages are also helpful for debugging errors in a new cell library.

If any description errors are found during compilation, error messages are issued that specify the location (line number) and nature of each error. The compiler aborts when the number of fatal description errors reaches a user-specifiable limit. This prevents voluminous output in the event of a repeating problem (such as an incorrect **!FORMAT** statement).

The result of a successful circuit compilation is that a binary representation is loaded into memory, ready for simulation. This representation can also be saved and later retrieved to avoid the need to recompile the same circuit description in future SIMIC sessions.

Many of the SIMIC commands require that the binary circuit representation be loaded when they are issued.

The **GET** run command is used to initiate circuit compilation and/or retrieve a previously-compiled description.

2.2.2 Circuit Compilation

2.2.2.1 Initiating Circuit Compilation

The **TYPE** keyword initiates compilation.

Therefore, the compilation options described below should either precede, or be placed in, the **GET** command containing the **TYPE** keyword.

Compilation is invoked with the **TYPE** (**TY**) keyword option of the **GET** (**GE**) run command. If the entire circuit is contained in a single file, and if the file's name is the default name (as specified in the **DEFINE FILE** run command) and its extension is **net**, then the basic command:

get type=<*main type name*>

is sufficient, where ***<main type name>*** is the name of the TYPE to be compiled. This TYPE is called the ***main type*** throughout this Guide.

If, however, the name of the file does not match the default name, or if the description is contained in more than one file, the **FILE** (**FI**) keyword must be used to direct SIMIC to the correct file(s):

get type=<*main type name*> file=<*list of files*>

where ***<list of files>*** is a list of all the files that contain the complete description. For example, if the description of the main type, **FULL_ADDER**, is distributed in the files **FAD** and **CMOSLIB**, (with default extension **NET**) the command:

```
get type=full_adder file=fad,cmoslib
```

would be used to compile the circuit description. With one exception[1], the

1. The exception arises, for example, when two or more macros (structural TYPEs) have been assigned the same type name. This is an error, since all macro names must be unique. In this case, SIMIC accepts the first type block that it encounters, as a result of the file list's ordering, and rejects all other identically-named type blocks, issuing an error message in the process.

order in which the files are listed is not important; what matters is that the list is complete.

The **!include** directive within a network description file directs SIMIC to other files required to complete the network description.

An alternative method of directing the SIMIC compiler to both files, in the above example, is to issue the command:

```
get type=full_adder file=fad
```

and have the following **!INCLUDE** statement in file **FAD**:

```
!INCLUDE cmoslib
```

SIMIC supports the ability to simulate multiple circuits within the same session. All that is necessary is to issue a **GET** command when ready to simulate the next circuit. Since all previously-issued commands were associated with the previous circuit, and have no relevance for the next one, SIMIC discards prior simulation options, and restores its initial defaults, whenever the **GET TYPE** command is issued.

If case-sensitivity is required, start the SIMIC session with the “s” switch before compilation:

simic –s

2.2.2.2 Specifying The Compiler’s Abort Limit

Since they do not conflict, compilation options may be freely mixed within the same **GET** command.

As previously mentioned, the SIMIC circuit compiler will abort the compilation process if the number of fatal errors reaches a predefined limit. This limit is defaulted to 20, but can be set to any level, or removed entirely. To set the limit to a new value, use the **STOP** (**STOP**) keyword option:

get stop=<*n*>

where ***<n>*** is an integer specifying the new limit. The **NO** prefix is used to remove the limit entirely:

```
no get stop:
```

Since use of the **TYPE** keyword initiates compilation, the **STOP** limit should be specified either in a separate **GET** statement prior to the **GET** command containing the **TYPE**, or in the same command:

```
get stop=15
get type=full-adder
```

or

```
get type=full-adder stop=15
```

The **NO** form of the command *must* be issued prior to the **GET TYPE** command.

2.2.2.3 Saving The Compiled Description In A File

After compilation, an image of the simulation structures can be saved into a file for quick retrieval in future simulations. This is beneficial for two reasons:

1. The compiler uses and then frees memory during the compilation process. This causes memory fragmentation to occur. If you have limited real memory for simulation, then avoiding the compilation process will cause more efficient utilization of memory during simulation.
2. Circuit restoration is significantly faster than circuit compilation.

By default, the compiled circuit description is not saved. The **SFILE** (**SF**) keyword may be used to override this default. The form:

```
get sfile:
```

implicitly instructs SIMIC to save the description in a file whose name is the default file name and whose extension is **rnt**, the default extension for compiled network descriptions.

In the second form:

get sfile=<*file name*>

SIMIC is directed to save the compiler output into the file named ***<file name>***.

The **NO** prefix can be used to cancel a previous request to save the compiled description:

```
no get sfile:
```

Since use of the **TYPE** keyword initiates compilation, **SFILE** should be specified either in a separate **GET** statement prior to the **GET** command containing the **TYPE**, or in the same command:

```
get sfile:
get type=full-adder
```

or

```
get type=full-adder sfile:
```

The **NO** form of the command *must* be issued prior to the **GET TYPE** command.

2.2.2.4 Obtaining A Readable Description Of The Flattened Circuit

During the compilation process, the circuit description is flattened, that is, expanded to an interconnection of built-in and user-defined primitives. An output file, called the **listing file**, can be optionally generated that contains a textual representation of the flattened circuit. If not explicitly requested, this file is not generated.

The **REPORT** (**REP**) keyword option specifies the amount of detail to be written to the listing file. Valid values for this option are **SYMBOLS** (**S**), and **ALL** (**A**). The SYMBOLS value specifies that only the names of the signals and parts are to be listed, and ALL specifies that in addition to these names, the topology and electrical attributes are to be listed as well. After a successful **GET**, the requested information is placed into the listing file specified by the **LFILE** (**LF**) keyword. If this keyword is omitted, or is specified as **LFILE:**, the listing file's default name is defined by the **DEFINE FILE** run command, and its default extension is **lst**. To override the default, the **LFILE** keyword should explicitly specify the file's name. For example:

```
define file=fad
get report=all
```

request that the listing file, **fad.lst**, contain signal name, part name, topology and electrical attributes, whereas:

```
define file=fad
get report=symbols lfile=fad1.out
```

requests that the listing file, **fad1.out**, contains only signal and part name information. If the listing file is not desired, after issuing the above commands, the previous requests can be cancelled by either of the following commands:

```
no get report:
no get lfile:
```

Since use of the **TYPE** keyword initiates compilation, the listing file options should be specified either in a separate **GET** statement prior to the **GET** command containing the **TYPE**, or in the same command:

```
get report=all lfile:
get type=full-adder
```

or

```
get type=full-adder report=all lfile:
```

The **NO** form of the command *must* be issued prior to the **GET TYPE** command.

2.2.3 Retrieving A Previously-Compiled Description

If a circuit description has been previously compiled, and the binary description was saved using the **SFILE** option with the **GET TYPE** command, then this description can be retrieved with the **RFILE** (**RF**) keyword option:

```
get rfile:
```

if the description was saved in a file having the default name and default file extension **rnt**, or

get rfile=<*file name*>

if the file's name need be explicitly specified.

The **TYPE** and **RFILE** keyword options are mutually exclusive and must not be entered as options in a single **GET** command. If both options are specified simultaneously, they are ignored and a warning message is issued.

As with the **GET TYPE** command, SIMIC discards prior simulation options, and restores its initial defaults, whenever the **GET RFILE** command is issued.

2.2.4 Selecting A Timing Table

Either of three tables (delay sets), corresponding to **TYPICAL**, **MINIMUM**, and **MAXIMUM** timing values may be specified for delay, loading and timing-check information. By default, the **TYPICAL** table is selected for simulation. To specify a table, use the **TIMING** (**TI**) keyword option:

get timing=<*table*>

where ***<table>*** is either **TYPICAL**, **MINUMUM**, or **MAXIMUM**.

The **timing** option may be used in conjunction with the **get type** or **get rfile** commands or as an independent command.

All timing specified in the network description is saved during circuit compilation, and thus can be selected at any time during a SIMIC session.

2.2.5 Backannotation

In addition to using the backannotation elements (**LOAD** and **DELAY**) in the network description, a separate file containing **LOAD** elements can be used to backannotate the circuit with loading. The loading specified by these elements is added to the loading already present at the specified nodes. The backannotation file is read with the **AFILE** (**AF**) keyword option:

```
get afile:
```

or

```
get afile=<file list>
```

The **afile** option can be used in conjunction with the **get type** or **get rfile** commands, or issued independently. Subsequent **get afile** commands will restore the loading specified in the network description before adding the new loading.

The backannotation file has the default extension **ann**, and consists of (optional) **SNL !FORMAT** directives, comments, remarks, and part statements instantiating SIMIC **LOAD** elements whose **OUTPUT-LOADS** (**OLOD**) keyword-fields specify the incremental net loading. For example, the following file assigns a load of 10 units to node **a** and 17 units to node **main.q**:

```
Remark= This adds loading to the signals
Remark= named 'a' and 'main.q'
!format p= t= olod=
a         load      10
main.q    load      17
```

Chapter 2.3 Input Stimuli

2.3.1 Introduction

After the **main type**'s compiled description has been loaded by the **`GET`** command, external stimuli must be defined and applied to its pins in order to exercise its functionality and verify its design. The stimuli are applied to the main type's input (unidirectional) and bus (bidirectional) pins. These ports are collectively called ***primary inputs*** (even though the busses will sometimes function as outputs, they must still be driven the remainder of the time from the external world), and the stimuli are called ***input stimuli***.

SIMIC supports three different modes of simulation that are specific to the circuit's basic operation and the reasons for performing the simulation. The simulation mode is determined by the input stimuli:

1. **Simulate-till-stable** – selected by applying **patterns**, this enforces Fundamental Mode operation.
2. **Time-based inputs** – selected by applying **waveforms**, this mode supports arbitrary primary input timing.
3. **Tester emulation –** selected by applying **timing generators**, this mode supports tester program debugging.

2.3.2 Types Of Input Stimuli

SIMIC supports three different types of input stimuli:

2.3.2.1 Patterns

If applicable, this stimulus mode is extremely helpful in trouble-shooting designs. For patterns, only the input values are specified, not their times of application. SIMIC applies each input pattern, simulates the circuit until its state becomes *stable* (internal activity ceases, and every element output state is consistent with its input and internal state), and only then applies the next input pattern. This mode of operation is also called Fundamental Mode, and many circuits are designed to operate exactly in this manner.

Patterns are useful for a number of reasons:

1. The circuit is <u>guaranteed not to lose Fundamental Mode operation</u>, since the effects of a subsequent input pattern cannot interact with events caused by the current pattern.

2. Simulation time is reset to 0 at the start of each pattern. Coupled with the option to output simulation results when the circuit state is stable, the circuit's <u>response time</u> for each input pattern is visible in the simulation output.
3. As a debugging aid, SIMIC always saves the state of the network at the previous stable point. Since a timing hazard must be caused by a single pattern, the user can interactively "replay" the hazard to obtain insights on the cause and ultimately the fix of the hazard. This dramatically <u>reduces circuit debugging time</u>. (Note that this is the only stimulus mode where the circuit is guaranteed to achieve stability for each input state; and therefore the only mode where the network state is guaranteed to be saved prior to any event of interest.)

2.3.2.2 Waveforms

For waveforms, both the input values and their times of application are specified. Each input state is then applied at its specified time, regardless of the stability of the circuit.

Waveforms should be used to test circuits whose input arrival times are unconstrained. These include asynchronous circuits, dynamic logic, frequency-sensitive circuits, retimers (e.g. UARTS), and non-synchronous pipelined architectures.

2.3.2.3 Timing Generators

Timing generator mode emulates modern automatic test equipment, thus allowing <u>test programs to be debugged using simulation techniques</u>. In this mode, stimuli are defined by input drive values, timing generators, and a master clock period, and outputs are strobed. The period, timing generator definitions and strobe placements may be dynamically changed during the simulation (time-set switching). Chapter 2.8 contains a full description of timing generators.

2.3.3 Test Numbers

In order to maintain a consistent reference, SIMIC assigns a **<u>test number</u>** to each distinct input state. The first input state applied is Test 1, the second is Test 2, and so on. Test numbers are incremented as follows:

1. patterns – whenever a new pattern is applied. Also **<u>time is reset to 0</u>**
2. waveforms – whenever an input changes state
3. timing generators – whenever a new test period is entered; thus, the test number is equivalent to the tester period number.

2.3.4 Specifying Stimuli As A Two-Step Process

Input stimuli are specified in a two-step process. In the first step, the patterns, waveforms, or timing generators are specified with the **`DEFINE`** run command. Then these defined sequences are applied to the proper inputs with the **`APPLY`** command.

This two step approach has the following advantages:

1. Stimuli may be defined hierarchically.
2. If the same sequence is to be applied to different inputs, then that sequence needs to be defined only once.
3. If more than one mode will be used to simulate the circuit, stimulus definitions in terms of patterns, waveforms, or timing generators can coexist without conflict, since only one type is applied for actual simulation.

2.3.5 Defining Input Stimuli

Primary input stimuli are defined using the **`DEFINE`** (**`DE`**) run command. The syntax is:

`define` *<pw><name>*`.`*<width><defaults>*`=` *<sequence>*

where:

- ***<pw>*** is either a **`P`** (for patterns) or **`W`** (for waveforms),
- ***<name>*** is a user-defined name for the input sequence, immediately following the ***<pw>*** designator; combined, the two entries form the sequence's complete name,
- ***<width>*** is the number of signals for which the input sequence is defined,
- ***<defaults>*** (optional) specifies default attributes of the stimulus sequence appearing on the right side of the equal sign, and
- ***<sequence>*** is the stimulus sequence being defined.

The optional ***<defaults>*** on the left side of the equal sign are:

.<duration>.<format>.<strength>

where:

- ***<duration>***, called the **default duration**, is

  for patterns, the *number of tests* to maintain each input state of ***<sequence>*** before applying the next one. If unspecified, each input state is maintained for one test (the default duration is 1)

<u>for waveforms</u>, the *amount of time* to maintain each input state of ***<sequence>*** before applying the next one. If unspecified, the default duration is 0, which means that all stimulus timing must be described in ***<sequence>***

- ***<format>*** is the default format (radix) of ***<sequence>*** (**BINARY**, **OCTAL**, **HEXADECIMAL**, **INTEGER**). If unspecified, the default is **BINARY**, and
- ***<strength>*** is the default strength of ***<sequence>*** (**POWER**, **DRIVING**, **RESISTIVE**, **FLOATING**). If unspecified, the default is **DRIVING**.

<format> and ***<strength>*** specifications may be abbreviated to any valid prefix, even one character. For example, **HEXADECIMAL** can be specified as: **H**, **HE**, **HEX**, **HEXA**, etc. These options are described in the *Stimulus Drive Strength* and *Stimulus Format* Sections of this Chapter.

Note that a single dot (.) separates each pair of adjacent specifications on the left side of the equal sign.

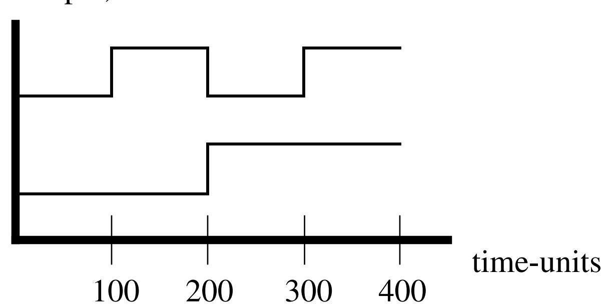
For example, the command:

```
define pall.3 = 000 001 010 011 100 101 110 111
```

defines a <u>pattern</u>, since the ***<pw>*** character is **P**. The pattern's name is **pall**, and its width specification, **3**, is separated from the name by a dot (.). Since there are no ***<duration>***, ***<format>***, or ***<strength>*** specifications following the width entry, the default duration of each stimulus state is one test, the radix of ***<sequence>*** is **BINARY**, and the stimuli have **DRIVING** strength.

The three-input binary sequence to be assigned the name **pall** (the ***<sequence>***) is specified to the right of the equal sign. This sequence consists of eight input states. In the first state, all inputs are 0. In the second state, the first two inputs are 0 and the third input is 1, etc.

As another example, the time-based stimuli shown below:



can be described as:

```
define wsample.2.100= 00 01 10 11
```

This statement specifies ***<pw>*** as **w** (a time-based waveform), ***<name>*** as **sample** (so the waveform's complete name is **wsample**), ***<width>*** as **2**, and ***<duration>*** as **`100`**. Since format and strength are not specified, the format is **`BINARY`** and the strength for each stimulus is **`DRIVING`**.

The names assigned to stimulus sequences must be unique. SIMIC does, however support stimulus redefinition; if a stimulus definition has the same name as a previously-read definition, it replaces the original one. The only restriction is that their ***<width>***s must be identical.

2.3.6 Selecting Stimuli For Simulation

After a stimulus sequence is defined and named by the **`DEFINE`** command, it must be attached to the primary inputs/busses of the circuit. This is done with the **`APPLY`** (**`AP`**) **`PATTERN`** (**`PA`**) run command. The syntax of this command is:

```
APPLY PATTERNS=<pwname> LIST=<signals> BEGIN=<n>
```

where:

- ***<pwname>*** is the name assigned to the sequence by a preceding **`DEFINE`** command, beginning with **`P`** if the sequence is a pattern or with **`W`** if the sequence is a waveform,
- ***<signals>*** is a list of the primary input/ bus pins to apply the stimuli to, and
- ***<n>*** is an offset (skew) from the current time or test for delaying application of the stimuli. If unspecified, the offset is 0.

The order of the primary signals in ***<signals>*** specifies the correspondence between bit positions in ***<pwname>*** and the signals. The first signal in ***<signals>*** will be associated with the first bit position in ***<pwname>***, and so on.

For example, using the above definition of **`pall`**:

```
apply patterns=pall list=c,b,a
```

associates the three-bit pattern named **`pall`** with the three primary signals **`c`**, **`b`**, and **`a`**, in that order. These three signals will therefore execute the following sequence, where the duration of each state is one test:

```
TEST: 1 2 3 4 5 6 7 8
   c: 0 0 0 0 1 1 1 1
   b: 0 0 1 1 0 0 1 1
   a: 0 1 0 1 0 1 0 1
```

Similarly, using the above definition of **`wsample`**:

```
apply patterns=wsample list=x,y
```

associates the two-bit waveform named **wsample** with the two primary signals **x** and **y**, in that order. These two signals will therefore execute the following sequence, where the duration of each state will be 100 time-units:

```
TIME: 0  100 200 300
   x: 0   0   1   1
   y: 0   1   0   1
```

The **apply** command's **list** keyword-field is necessary except for the special case of an input sequence whose width and ordering exactly match the number and ordering of primary inputs of the main type's **type statement**.

If the stimulus width is equal to the total number of primary inputs and busses, and the pattern values are ordered as inputs followed by busses, and the input and bus values are themselves ordered as they appear in the circuit's main type statement, then the **LIST** (**LI**) keyword-field may be omitted.

For example, if the input keyword-field of the main type's TYPE statement is **I=X,Y**, and there are no primary busses, then the **LIST=X,Y** keyword-field in the above **APPLY** command is optional, since this would be the default association of primary inputs and stimuli.

Subsequent **APPLY** commands override previous **APPLY** commands for common primary signals. For example, if the command:

```
apply patterns=pat1 list=a,b,c
```

is erroneously issued, and the intended pattern for these signals is **pat2**, the subsequent command:

```
apply patterns=pat2 list=a,b,c
```

overrides the previous error and applies **pat2**. Primary input grouping need not be identical in **APPLY** commands. For example, if **pat3** has a width of 1, and if the command:

```
apply patterns=pat3 list=c
```

is subsequently issued, then the first two bit positions of pattern **pat2** will be applied to inputs **a** and **b**, and **pat3** will be applied to input **c**.

The **BEGIN** (**BE**) keyword is optional and is used for the following:

1. In waveforms, **BEGIN** can skew one waveform with respect with another. This simplifies testing the circuit's sensitivity to input skew.
2. The **BEGIN** option can also be used to apply "patches" (positioned stimuli) to the inputs in any mode. The patch is applied at the test (for patterns) or time (for waveforms) specified by the **BEGIN** keyword-field.

2.3.7 DO Loops For Repetitive Sequences

Any repeating sequence (or subsequence) of stimulus states may be specified as a loop. The form of a loop is:

```
do <count> (<sequence>)
```

where:

- ***<count>*** is the number of times to repeat the loop, and
- ***<sequence>*** is the sequence to be repeated.

Optional whitespace may precede or follow the repetition factor, ***<count>***.

Loops can be nested within loops. There is no practical limit to the level of nesting.

For example, the command:

```
define pr.1 = 0 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1
```

defines a pattern sequence for one primary signal. Its default duration is one test. This sequence can also be expressed as:

```
define pr.1 = do 2 (0 1 0 1 1 1 1 1)
```

or as:

```
define pr.1 = do 2 ( do 2 (0 1) do 4 (1))
```

or even as:

```
define pr.1 = do 2 ( do 2 (0 1) do 2(do 2(1)))
```

2.3.8 Grouping

Stimuli may be defined for subgroups of the primary signals, as desired, to facilitate stimulus definition. Inputs that would typically be partitioned are master clear and/or preset, clocks, and synchronous data lines. For each partition there must be an associated **APPLY** command. For example, consider the eight pattern sequence, **pall**, illustrated above:

The **list=c,b,a** option can be omitted here, if the main type had only three inputs, ordered: **c**,**b**,**a**

```
define pall.3 = 000 001 010 011 100 101 110 111
apply pattern=pall list=c,b,a
```

Another way to define and apply this sequence is:

```
define pcb.2.2= 00 01 10 11
define pa.1= do 4 (0 1)
apply pattern=pcb list=c,b
apply pattern=pa list=a
```

Here, pattern **pcb** is defined and applied to signals **c** and **b**, and pattern **pa** is define and applied to signal **a**. Note that the default duration of **pcb** is **2**, so this pattern's sequence is actually:

```
00 00 01 01 10 10 11 11
```

A third method of defining and applying these patterns is:

```
define pc.1.4= 0 1
define pb.1.2 = do 2 (0 1)
define pa.1= do 4 (0 1)
apply pattern=pc list=c
apply pattern=pb list=b
apply pattern=pa list=a
```

Note that patterns **pc** and **pb** are equivalent to:

```
define pcequiv.1 = 0 0 0 0 1 1 1 1
define pbequiv.1 = do 2 (0 0 1 1)
```

If the applied groups do not all end at the same test, then the last stimulus value of each group will be maintained until all groups are finished. For example, suppose that circuit has two inputs, **clock** and **reset**, and the input sequence must:

1. Apply the reset signal, which is active-high.
2. Remove the reset.
3. Clock the device 64 times.

This could be accomplished by:

```
define preset.1 = 1 0
define pclock.1 = 0 0 do 64 (1 0)
apply pa=preset li=reset
apply pa=pclock li=clock
```

in which the last 128 patterns of **preset** are omitted. During simulation, SIMIC will maintains the last value of **preset**, **0**, while the **do** loop of **pclock** is being expanded.

2.3.9 Stimulus Hierarchy

Stimulus definitions may reference other stimulus definitions as long as the **width** of both definitions are the same. This is accomplished by placing the referenced sequence's name at the proper location in the sequence being defined. There is no practical limit on the level of nested stimulus definitions.

For example, the pattern:

```
define pclock.1 = 0 0 do 64 (1 0)
```

could also be defined as:

```
define pfirst2.1.2 = 0
define pcycles.1 = do 64 (1 0)
define pclock.1 = pfirst2 pcycles
```

Here, pattern **pclock** contains two subsequences, the first being **pfirst2** and the second **pcycles**. During simulation, SIMIC expands **pclock** exactly in this order. **pfirst2** is expanded first; it consists of a single **0** level whose duration is two tests. After **pfirst2** is completed, expansion of **pcycles** begins (starting at Test 3).

An alternative hierarchical definition of **pclock** might be:

```
define pcycles.1 = 1 0
define pclock.1 = 0 0 do 64 (pcycles)
```

Primitive values are the symbols that directly represent logic levels.

Sequence entries that directly represent logic levels (e.g., **0**, **1**) are called **primitive values**. Thus, every sequence definition may contain a mixture of primitive values and hierarchical references. In general, a hierarchical reference can be placed wherever a primitive value can be placed.

Default duration, format, and **strength** do **not** apply to hierarchically-referenced sequences.

However, there is one significant difference between primitive values and hierarchical references; the default fields of a **DEFINE** command (***<duration>***, ***<format>***, ***<strength>***) *only apply to its primitive values*. Thus, the **DEFINE** command of last example above could have been:

```
define pclock.1.2 = 0 do 64 (pcycles)
```

Here, the default duration of **2** only applies to the primitive value **0**. The default duration of primitive values in **PCYCLES** is determined by *its* **DEFINE** command.

2.3.10 Stimulus Drive Strength

If a primary input is only connected to inputs of unidirectional elements, then its drive strength is not very important. The drive strength of an external source connected to a primary *bus*, however, may be crucial for proper operation. A primary bus can be driven either by an external source or by a driver inside the circuit being simulated. Sometimes, it may be driven by both sources simultaneously, whether by design (e.g., a strong reset) or by error (e.g., a data bus that should have at most one driver at any time).

Thus, it is sometimes necessary to specify the drive strength of primary stimuli. The *Defining Input Stimuli* Section of this Chapter introduced the **DEFINE** command's default ***<strength>*** option. This option specifies the drive strength of *primitive values* in the sequence being defined. One of four strengths may be specified, corresponding to the supported drive strengths for gate-level components:

POWER, **DRIVING**, **RESISTIVE**, **and FLOATING**.

Any prefix of these strength designations is a valid specification. If this option is unspecified, the default drive strength is DRIVING.

The default strength specification does not affect *all* primitive values. Some primitive values have an associated strength as well as level, and the associated strength prevails over the default strength specification. For example, the primitive entry **Z** (or **z**) represents “unknown level at floating strength”. The **DEFINE** commands:

```
define wseq_d.1.50 = 0 1 x z
define wseq_r.1.50.r = 0 1 x z
```

both define a sequence of values consisting of **0** for 50 time-units, **1** for 50 time-units, **X** (representing “unknown; either **0** or **1**”) for 50 time-units, and **Z**. Since no default strength is specified for **WSEQ_D**, its **0**, **1**, and **X** values will be applied at DRIVING strength. Since a default strength of **R** is specified for **WSEQ_R**, its **0**, **1**, and **X** values will be applied at RESISTIVE strength. The last state of both sequences will be the same floating-unknown value, **Z**.

In addition to **Z**, other primitive value symbols represent drive strengths as well as binary logic levels. These are also the symbols that SIMIC uses to report signal states. The table below shows these symbols and the combined level/strength values they represent. Associated mnemonics are in italics:

	POWER	RESISTIVE	FLOATING
Logic-0	G (*ground*)	L (*low*)	D (*discharged*)
Logic-1	V (*vdd*)	H (*high*)	C (*charged*)
Unknown	S (*short*)	Y	Z

Table 2.3-1 Symbols Representing Combined Level/Strength

2.3.11 Stimulus Format

2.3.11.1 Default Format Specification

The *Defining Input Stimuli* Section of this Chapter introduced the **DEFINE** command’s default ***<format>*** option. This option specifies the default format of *primitive values* in the sequence being defined. One of four formats may be specified:

BINARY, **OCTAL**, **HEXADECIMAL**, **INTEGER**.

Any prefix of these format designations is a valid specification. If this option is unspecified, the default format is BINARY.

2.3.11.2 Format Descriptions

The following summarizes the primitive values representing stimulus levels for each format. The corresponding drive strength is specified by the default ***<strength>*** option:

<u>BINARY FORMAT</u>

Sequence specification using the BINARY format has been illustrated throughout this chapter. Each stimulus state, consisting of ***<width>*** levels, is represented as ***<width>*** binary levels. Logic levels 0 and 1 are represented by the identical symbols, **`0`** and **`1`**. (Additionally, as described in the *Stimulus Drive Strength* Section of this Chapter, nine symbols are used to represent combined level/strength states.)

<u>OCTAL FORMAT</u>

Each stimulus state, consisting of ***<width>*** levels, is represented as a right-justified octal number. Levels 000 through 111 are represented by the corresponding octal symbols **`0`** through **`7`**.

<u>HEXADECIMAL FORMAT</u>

Each stimulus state, consisting of ***<width>*** levels, is represented as a right-justified hexadecimal number. Levels 0000 through 1111 are represented by the corresponding hexadecimal symbols **`0`** through **`F`** (or **`f`**).

<u>INTEGER FORMAT</u>

Each stimulus state, consisting of ***<width>*** levels, is represented as either a positive or negative integer. The integers represent the binary number formed when the leftmost level is used as the most significant bit. Negative integers specify the 2's complement binary representation.

The only format restricted in size is INTEGER. This format can represent, at most, 32 signal levels. There are no restrictions for the other formats.

Additionally, some symbols are used for all formats. They specify values for a group of signals, where the group is:

- <u>a single signal</u> for binary format
- <u>all signals of an octal digit</u> for octal format
- <u>all signals of hexadecimal digit</u> for hexadecimal format
- <u>all ***<width>*** signals of the stimulus sequence</u> for integer format.

These common symbols are:

- **`X`** - specifies that all signals of the group are X (could be either 0 or 1)

- **Z** - specifies that the all signals of the group have floating unknown values (this is the only primitive value that also specifies drive strength for all formats)
- **I** - specifies that all signals of the group be generated by inverting (bit-wise complementation) their *previous* levels
- **N** - specifies that all signals of the group be generated by copying their *previous* levels.

For example, the following **DEFINE** commands are equivalent:

```
define pabc.5 = 00000 11111 01101 zzzzz xxxxx
define pabc.5 = 00000 iiiii innin zzzzz xxxxx
define pabc.5.oct = 00 37 15 zz xx
define pabc.5.oct = 00 i7 15 zz xx
define pabc.5.hex = 00 1f 0d zz xx
define pabc.5.hex = 00 1f id zz xx
define pabc.5.int = 0 31 13 z x
define pabc.5.int = 0 -1 13 z x
define pabc.5.int = 0  i 13 z x
```

2.3.11.3 Radix Escapes

The radix may be explicitly switched for specific input states of the stimulus definition. This is done by prefixing each input state with a radix escape character from the table below:

Escape	Selected Radix
^	BINARY
*	OCTAL
#	HEXADECIMAL
%	INTEGER

Each radix escape character only affects a single input state.

Radix escape is useful when signals grouped within a digit must be assigned incompatible values. For example:

```
define pbus.4.hex= 0 ^01XX %-1 Z *0Z
```

defines the following patterns:

```
0000 01XX 1111 ZZZZ 0ZZZ
```

Here, the second and fifth input states could not be specified in hexadecimal format, since **X** or **Z** would set all *four* bits to the respective values.

Changing radix may also facilitate stimulus description. For example, suppose an 8-bit ALU performs arithmetic operations on its data inputs, **`a`** and **`b`**, when its 2-bit control input, **`func`**, is 0 or 1, and logical operations on the inputs when FUNC is 2 or 3. Stimulus description would be simplified if the data inputs were specified as integers for the arithmetic operations and as hexadecimal (or binary or octal) for the logical operations. For example, the following commands define a few (non-comprehensive) tests for the ALU:

```
define pfunc.2.int=  0    1    2    3
define pa.8.int=    -2  255  #0f  #50
define pb.8.int=     1  255  #f5  #0a
```

2.3.11.4 Guidelines for Entering Proper Values

The following guidelines should be followed for specifying stimulus values:

1. For binary, octal, and hexadecimal formats:
   a. exactly the right number of digits must be used to represent each stimulus state value
   b. spacing between different stimulus state values is optional, but a space should *not* appear within (break up) a stimulus state value for octal and hexadecimal formats
2. For integer formats, whitespace or commas *must* appear between stimulus state values
3. For all formats, the value specified for each stimulus state must be consistent with the number of primary signals in the sequence
4. Radix escapes must specify a complete stimulus state value (i.e., specify the values of *all* primary signals associated with the stimulus).

Simply stated, (1a) specifies that leading zeros of octal and hexadecimal digits cannot be omitted, and no extra digits can be added to a stimulus state specification, since the extra digits will be interpreted as belonging to the *next* stimulus state. The latter possibility is due to (1b), which states that stimulus specification for these radices is format-free between state values.

More formally, (1a) requires that each stimulus state specification consist of exactly

ceiling(***<width>***/$\log_2$(*radix*))

digits, where the *ceiling* function is "the smallest integer greater than, or equal to, its argument", and *radix* is 2, 8, or 16 for BINARY, OCTAL, or HEXADECIMAL formats, respectively. Thus, the correct number of digits per stimulus state value specification must be

<width>, *ceiling*(***<width>***/3), or *ceiling*(***<width>***/4),

respectively, for the three formats.

Guideline (2) is fairly obvious; since integers can contain a varying number of digits, *something* has to delimit them.

Guideline (3) states that the stimulus state value cannot exceed

$(2^{<width>-1}-1)$.

For negative integers, this value cannot be less than

$(-2^{<width>-1})$.

Any value specified outside these limits is out of range for the given number of primary signals. If the correct number of digits are used, but the specified value is out of range, SIMIC discards the extraneous most-significant bits, issues a warning message, and accepts the truncated value.

Guideline (4) states that, within a single stimulus state, it is an error to define some primary signals in one radix and the remaining signals in another radix.

2.3.12 Stimulus Positioning

In addition to the default duration option, ***<duration>***, on the left side of the **`DEFINE`** command, stimulus states may be positioned using two constructs on the right side (within the sequence being defined). These constructs are **absolute positions** and **explicit durations**. They eliminate the need to repeat levels that remain constant for long intervals, and allow positioning at points that are not multiples of the default duration.

To demonstrate the features of positioning the sequence of patterns described previously, **`pall`**, will be used here:

```
TEST: 1 2 3 4 5 6 7 8
   c: 0 0 0 0 1 1 1 1
   b: 0 0 1 1 0 0 1 1
   a: 0 1 0 1 0 1 0 1
```

The patterns for **`a`**, **`b`**, and **`c`** be defined separately as:

```
define pa.1= 0 1 0 1 0 1 0 1
define pb.1= 0 0 1 1 0 0 1 1
define pc.1= 0 0 0 0 1 1 1 1
```

2.3.12.1 Default Positioning

Briefly reviewing, a default duration, ***<duration>***, can be specified in the definition of a stimulus sequence. If unspecified, the default duration is implicitly 1 for patterns and 0 for waveforms. Using a default durations, the pattern definitions for **`b`** and **`c`** can be simplified:

```
define pb.1.2= 0 1 0 1
define pc.1.4= 0 1
```

As can be seen, the default duration simplifies description when a stimulus sequence contains periodic changes. However, repetitive levels must still be entered for nonperiodic changes:

```
define wx.1.10 = 0 1 0 0 1 1 0 0 0 1 1 1 0
```

2.3.12.2 Absolute Positioning

Absolute positioning places a primitive value, do loop, or hierarchical reference at an absolute position with respect to the beginning of the sequence. Absolute positioning is specified as an at-sign (**@**) followed by a position value. The immediately *following* primitive value, do loop, or hierarchical reference will be placed at the specified position.

Using absolute positioning, the definitions for **b** and **c** above might be:

```
define pb.1= 0 @3 1 @5 0 @7 1
define pc.1= 0 @5 1
```

This definition of **pb** places a 0 at Test 1, a 1 at Test 3, a 0 at Test 5, and a 1 at Test 7. Similarly, the definition of **pc** places a 0 at Test 1 and a 1 at Test 5.

Similarly, the definition of **wx** using absolute positioning could be:

```
define wx.1.10 = 0 1 0 @40 1 @60 0 @90 1 @120 0
```

As mentioned above, absolute positioning is relative to the start of the sequence. If the sequence is within a local loop, then the most immediate left parenthesis marks the sequence's start. For example, in:

```
define py.1= 0 @5 do 2 (1 @3 0)
```

the absolute position specification, @3, is with respect to the left parenthesis, not the beginning of the entire sequence (which @5 references). This definition expands to:

```
define py.1= 0000 110 110
```

2.3.12.3 Explicit Duration

Explicit durations specify how long to hold the *previous* stimulus state, prior to initiating the next one. The previous state might have resulted from a primitive value within the current definition or from a hierarchical reference. Explicit duration is specified as an ampersand (**&**) followed by the duration value. The previous stimulus state will be held for the specified duration.

Again, **b** and **c**, could be defined in terms of explicit durations as:

```
define pb.1= 0 &2 1 &2 0 &2 1
define pc.1= 0 &4 1
```

Here, in **`pb`**, 0 is placed at Test 1 and held for 2 tests, then 1 is placed and held for 2 tests, etc. Similarly, the 0 in **`pc`** is held for 4 tests until the 1 is applied at Test 5.

2.3.12.4 Positioning Precedence

The positioning methods described above may be freely mixed. The rule is: "do whatever is easiest". If there is a conflict between positioning specifications, SIMIC resolves them by the following order of precedence (highest to lowest):

1. Absolute positioning (@ specification).
2. Explicit duration (& specification).
3. Default duration (***<duration>*** specification).

Chapter 2.4 Simulation Output

2.4.1 Overview

SIMIC creates several simulation output files. The textual simulation reports described here are organized in a tabular format whose content and form are, to a large extent, user-controllable. The output's destination can be the terminal (or log file in batch mode), via the **PRINT** (**PR**) run command, and/or a file, via the **WRITE** (**WR**) run command.

Since options common to the two commands are identical, sample usage is illustrated for only one of the commands, **PRINT**. Substituting the command verb "WRITE" invokes the identical option for this command. Options specific to only one of the commands are explicitly stated.

Even though the **PRINT** and **WRITE** options are identical, the two commands are totally independent. The signals reported, the output format, and the report times are unique to each command; nothing specified for one applies to the other.

Each section of this chapter describes an individual **PRINT** or **WRITE** option and illustrates its usage. While isolated here for descriptive purposes, any of these options may be combined in the same command, if desired.

2.4.2 Organization Of The Output

Figure 2.4-1 illustrates a sample file created by the **WRITE** command. In general, the output generated by the **PRINT** or **WRITE** command contains:

1. A time/date/version stamp – This is only for **WRITE** output. It consists of a single line **REMARK** that contains the time, date and SIMIC version that created the file.
2. A simulation options header – This consists of a number of **REMARK** lines that describe the options selected for simulation.
3. A signal name header – This consists of a number of COMMENT lines that contain the names of the signals to be reported, as requested by the user, in vertical columns directly above their respective simulation values.
4. The simulation output – Contains a record for each time that output occurs during simulation. Each record contains the current simulation time and test, separated by a 'T' and suffixed with a ":", followed by the current values of the requested signals.

```
Remark= 'Write' Created by SIMIC 1.00.00 on 4/23/91 14:37:53

Remark= Options: (Fault Free simulation)
Remark=   Pattern stimuli, Near Filter, Spike Propagation
Remark=   Stable Before Decay, Dynamic Delay

C=                  INNNN
C=                  N1234
C=                  I
C=                  T

        0 T      1: 0ZZZZ
        0 T      2: 11ZZZ
        1 T      2: 11HZH
        2 T      2: 11HHH
        0 T      3: 0CCCC
        5 T      3: 0ZZZZ
        0 T      4: 11ZZZ
```

Figure 2.4-1 Sample File Created by the WRITE Command

2.4.3 Specifying The File Name For The WRITE Command

By default, the extension of the file created by the **`WRITE`** command is **wrt**, and its name is the default name specified by the **`DEFINE FILE`** command. The **`FILE (FI)`** keyword can be used to explicitly specify this file's name:

`WRITE FILE=`<*file name*>

where ***<file name>*** is the name of the file to be written to.

2.4.4 Specifying *What* To Output

2.4.4.1 Selecting Signals to Output

Signals are selected with the **`LIST`** (**`LI`**) keyword-field. Two forms of this keyword are supported.

The first, more common, form is:

`PRINT LIST=`<*signals and format options*>

This form specifies the signals (and formatting options) that should be output. A signal can be specified more than once. For example:

`PRINT LIST=a,b,c,a`

would output the signals **`a`**, **`b`**, **`c`**, and then **`a`** again in the tabular output.

The options for formatting the table are:

1. Inserting one or more blank vertical columns between signals. This is accomplished by entering an asterisk (**`*`**) for each blank column.
2. Forcing a new row in the table. This is accomplished by entering a pound sign (**`#`**) at the desired point. Note that if more than one **`#`** is used consecutively, then a blank row will be output.

Commas are optional before or after **`*`** or **`#`**.

For example:

```
PRINT LIST=u,v,*w**x#y
```

will cause the simulation output to consist of the value of signal **`u`**, followed by the value of **`v`**, followed by a blank column, followed by the value of **`w`**, followed by two blank columns, followed by the value of **`x`**. The value of signal **`y`** will be output at the first position of the next line.

Like most other keywords, the **`LIST`** keyword is "sticky"; any signal specified will continue to be reported until explicitly removed. For example, if the command:

```
PRINT LIST=a,b,c,a
```

is followed by:

```
PRINT LIST=d*e
```

then, the two commands are equivalent to the single command:

```
PRINT LIST=a,b,c,a,d*e
```

Removing signals from the list is accomplished with the **no** prefix. For example:

```
NO PRINT LIST=a,b
```

causes the *first* specification of signal **`a`**, and signal **`b`**, to be removed, and its cumulative effect would be equivalent the single command:

```
PRINT LIST=c,a,d*e
```

The second form of the **`LIST`** keyword is:

```
PRINT LIST:
```

which specifies that SIMIC should output the values of *all* signals, each time output is requested. Since the amount of data could be voluminous, SIMIC generates a **dump format**, which reduces the output by:

1. Not printing the names of the signals in the header. Instead, the signals are arranged alphanumerically, according to their order in the **`SYMBOL`** section of the listing file, which can be created during circuit compilation.
2. Displaying the signals in groups of 5 with an intervening space. This simplifies coordinating the values with the corresponding signal positions.

This form is rarely used, but it can be helpful when tracing an X value to it's source.

Using the **LIST:** form with the **no** prefix:

```
NO PRINT LIST:
```

cancels reporting of all signal values.

2.4.4.2 Signal Specification Options

As described above, the **LIST** keyword accepts a list of signals. This is also true for the **LIST** keyword associated with other commands. Except where noted in this Guide, the **LIST** keyword of each command will accept:

1. Signal names
2. Meta-words specifying primary signal values:
   a. **&INPUTS** – specifies "all primary inputs"
   b. **&OUTPUTS** – specifies "all primary inputs"
   c. **&BUSSES** – specifies "all primary busses"
   d. **&BUSINS** – specifies "the stimulus values at all primary busses" which may differ from the values of &BUSSES (the latter being the wire-tied result of primary stimulus values and internally-driven values)
3. Wildcard specification of the form ***<prefix>***(), which represents "all signal names beginning with ***<prefix>*"** (for example, **A.B.()** specifies "all signals whose names begin with **A.B.**")
4. Factored specification of the form ***<prefix>(<suffix1>, …, <suffixn>)***, which represents the signals
   <prefix><suffix1>, …, <prefix><suffixn>
   (for example, **ab(cd, ef, gh)** *specifies* **abcd***,* **abef***, and* **abgh***)*
5. **Vector aliases** defined in a **DEFINE** command (see *Specifying Signal Groups and Output Radix Format* in this chapter)
6. Alternative specification of each signal in the form ***<partname>.<n>***, where ***<partname>*** is the instance name of the part generating the signal, and ***<n>*** is the signal's output number (e.g., if the signal is the fourth output of this element, then ***<n>*** would be 4*)*.

2.4.4.3 Controlling Column Width

SIMIC, by default, assumes an 80 column output device. If the space required to output the requested signal values (including blank columns) exceeds this limit, then SIMIC will output the signals in groups of 5 with one intervening space. If a 132 column output device is available, the column limit can be changed to 132 with the **EXPAND** (**EX**) keyword:

```
PRINT EXPAND:
```

As described previously, using the pound sign (**#**) to force multi-line tabular format is another way to stay within the set column limit. To defeat all column checking, specify the value **INFINITE** (**I**):

```
PRINT EXPAND=INFINITE
```

2.4.4.4 Suppressing Header Output

The simulation header, consisting of the simulation options and signal names, can be suppressed with the **HEADER** (**HE**) keyword option. This may be desirable during an interactive debugging session with short, multiple runs. It helps reduces the size of the output. The command:

```
NO PRINT HEADER:
```

disables the header output, and the command:

```
PRINT HEADER:
```

enables it.

2.4.4.5 Suppressing Test Number

When waveform stimuli are used, the test number field of the simulation output can be suppressed to simplify comparisons with other simulator output. This is done with the **TNUM** (**TN**) keyword option:

```
NO PRINT TNUM:
```

The test field can be restored with the command:

```
PRINT TNUM:
```

If test information is suppressed, each output line will contain the time, followed by a colon, followed by the requested signal values.

2.4.5 Specifying *When* to Output

The keyword options described in this section control when SIMIC should output simulation values. They are independent of each other; enabling one option does not disable another. Like many other options, they are "sticky"; once specified, the options remain in effect until explicitly changed.

2.4.5.1 Requesting Output at Stable Points

The default output operation for SIMIC is to output the requested simulation values each time the circuit becomes stable. The **PSTEP** (**PS**) keyword option can be used to control frequency of this output, or inhibit it. The following command specifies that output at every **<s>**-*th* stable point, where **<s>** is an integer:

```
PRINT PSTEP=<s>
```

To turn off all output based on attaining a stable state:

```
NO PRINT PSTEP:
```

2.4.5.2 Requesting Time-Periodic Output

Output can be requested at specified time intervals using the **TSTEP** (**TS**) keyword. The command:

```
PRINT TSTEP=<t>
```

will cause an output to occur every ***<t>*** simulation time-units, where ***<t>*** is an integer.

Once the circuit state has stabilized, output is suspended until the next input event occurs to eliminate repetitive identical reports. Output suspension can be disabled by prefixing the time-step specification with a plus (“+”) sign. For example:

```
PRINT TSTEP=+100
```

produces output every 100 time-units, even when the circuit state is stable.

Once enabled, the **no** prefix can be used to suppress time-periodic output:

```
NO PRINT TSTEP:
```

It is also possible to skew (offset) the first **TSTEP** output. This is done with the **BEGIN** (**BE**) keyword option. This command has the form:

```
PRINT BEGIN=<b>
```

where ***<b>*** is an integer specifying the time offset to the first output. This option can useful when emulating a point strobe with waveform or timing generator stimuli.

To remove the offset, a value of 0 is specified:

```
PRINT BEGIN=0
```

2.4.5.3 Requesting Output Based On Activity

One of the most commonly used options is to select signals that trigger simulation output. The only restriction is that the trigger signals must be among those chosen for output.

Activity-based output is enabled with the **CHANGE** (**CH**) keyword option. The command:

```
PRINT CHANGE:
```

specifies that output should occur whenever any signal on the current output list changes state. Particular trigger signals can be specified with the command:

```
PRINT CHANGE=<signal list>
```

where ***<signal list>*** is a list of the triggering signals. Subsequently, the command:

```
NO PRINT CHANGE:
```

would disable all activity-based output, and

NO PRINT CHANGE=<*signal list*>

would inhibit the specified signals in ***<signal list>*** from functioning as triggers.

2.4.5.4 Restricting the Output to Specified Tests/Time

Output can be restricted to certain tests (for patterns) or time (for waveforms) intervals with the **`PRANGE`** (**`PR`**) keyword option. **`PRANGE`** is described in *Restricting Simulation Options to a Specified Simulation Interval* in Chapter 2.6.

2.4.6 Controlling Signal Value Representation

2.4.6.1 The Output Character Set

By default SIMIC uses a 15-character representation for the possible combinations of levels and strengths. These characters, and their associated signal states, are shown in the following table:

Strengths	Level 0	Level 1	Level X
POWER	G	V	S
DRIVING	0	1	X
RESISTIVE	L	H	Y
FLOATING	D	C	Z
UNKNOWN	F	T	U

2.4.6.2 Suppressing Signal Strength in the Output

It is sometimes distracting to see signal values in the simulation output described with the 15-character representation. The **`VALUES`** (**`VA`**) keyword directs SIMIC to represent signal values in a 4-character (**`0`**, **`1`**, **`X`**, **`Z`**) representation, ignoring all strength information except for floating-unknown (**`Z`**). The values can be either **`STRENGTHS`** (**`S`**) or **`LEVELS`** (**`L`**), corresponding to 15-character or 4-character, respectively. The command:

```
PRINT VALUES=LEVELS
```

selects the 4-character representation, while the command:

```
PRINT VALUES=STRENGTHS
```

selects the 15-character representation.

2.4.6.3 Specifying Signal Groups and Output Radix Format

In many instances, it is easier to read the output if a display radix other than binary (**LEVEL**) is used. If signals are grouped into arrays (vectors) in the network description, then the vector can be displayed in its declared radix format (in the SNL **%DECLARE** statement, see *Signal Arrays* in Chapter 1.2). This is accomplished by specifying the "root" name (the name without the dimension specification) at the desired position in the output list. For example, consider this partial SNL description:

```
T=alu I=a,b,func O=ovl,out
%DECLARE INT=a[0:7],b[0:7],func[0:1],out[0:7]
<part statements have been removed>
```

If the inputs and outputs (with intervening blank columns) are to be printed in their declared integer format, the following command can be used:

```
PRINT LIST=a*b*func**ovl*out
```

Sometimes it may be desirable to output a group of signals that are not declared as a vector, or to change the radix format of a declared vector, or even to reverse or modify the array order of a declared vector. All of these functions are handled by defining a **vector alias** with the **DEFINE** command. The format for this command is:

DEFINE V*<name>*.*<format>*=*<list of signals>*

where ***<name>*** is a user-supplied alias for the vector, and ***<format>*** is one of the following formats:

- **LEVEL** -- individual levels for each bit.
- **OCTAL** -- octal representation.
- **HEXADECIMAL** -- hexadecimal representation.
- **INTEGER1** (**INT1**) -- One's complement representation.
- **INTEGER2** -- Two's complement representation.
- **POSINTEGER** -- Positive integer representation.

Except for the one's complement specification, which can only be **INTEGER1** or **INT1**, any valid specification prefix is sufficient.

For example, to display **a** and **b** together in **HEXADECIMAL** format, and display **func** as an integer in the reverse array order, the following commands can be used:

```
DEFINE VAB.HEX=a,b
DEFINE VFUNC.INT=func[1:0]
```

```
PRINT LIST=vab*vfunc**a*b*vfunc**ovl*out
```

When **vab** and **vfunc** are displayed in the signal name header, the "**v**" prefix is replaced with an asterisk (*****). This is a reminder that the displayed signal is not an original signal, but is actually an alias generated at run time.

Once a vector alias is defined, it (like declared arrays) may also be used as a value for any keyword that processes a list of signals. In this case, the root name is synonymous with its associated list of signals.

For Octal and Hexadecimal formats, if all signal in the radix group are tristated (Z), then a Z will be displayed in that group's position. Otherwise, if any signal is unknown (X) or tristated (Z), then an X will be displayed in that groups position. For the integer formats, if all signals in the array are Z, then a Z is displayed. Otherwise, if any signal is X or Z, then an X is displayed.

2.4.6.4 Querying For Current Selected Options

SIMIC can be queried to report which options have been currently selected, which signals are currently selected for output, and which signals will trigger output when they change value, by issuing the command:

```
?WRITE
```

(**?WR**) for the write options, and:

```
?PRINT
```

(**?PR**) for the print options.

Chapter 2.5 Simulation Options

2.5.1 Overview

SIMIC is a very comprehensive simulator, with many user-controlled options. This chapter explains the simulation options summarized in the options banner, which is generated at the start of PRINT or WRITE output (see Chapter 2.4). Where necessary, it also indicates the chapters of this Guide that contain more detailed descriptions of these options.

A typical options banner might appear as follows:

```
Remark= Options: (Fault Free Simulation)
Remark=   Pattern Stimuli, Near Filter, Spike Propagation
Remark=   Stable After Decay, Dynamic Delay
```

This banner summarizes some of the main options chosen for this simulation. In addition, it is always possible to "query" SIMIC for a more detailed report of selected global options with the **?DEFINE** (**?DE**) command:

```
?DEFINE
```

For example:

```
>>: ?define
Global Definitions:
  Default File Name = 'noname'
  Resistive Strength to Depth value = 3
  Oscillation Limit = 10
  Warn Message Limit = 10
  Potential detect drop limit = Infinite
  Pulse window multiplier = 3
  Near window multiplier = 2
  X-address limit = 4
  Dynamic modification of delay = Yes
  Propagate Spikes = Yes
  Stability = After Decays
```

2.5.2 Fault Free Simulation/Fault Simulation

Fault-free simulation does not mean that the circuit is error-free; it means that SIMIC is not performing fault simulation. Fault-free simulation, sometimes called "true-value" or "good-logic" simulation, is the simulation mode for verifying circuit timing and functionality; it is the mode that has been discussed so far in this Guide. In contrast, fault simulation introduces logic faults and grades the input stimuli's ability to detect these faults. Hence, fault simulation verifies the effectiveness of test stimuli. As long as the **`FAULT`** command options are not activated, simulation will be in fault-free mode.

2.5.3 Pattern Stimuli/Waveform Stimuli/Timing Generators

This options banner entry reports the applied input stimulus mode, as discussed in Chapter 2.3. This choice determines the interpretation of the time and test fields in the simulation output. Again, pattern mode is the most powerful for circuit analysis and debugging, if applicable. Timing generator mode, by emulating the tester environment, is the most applicable mode for defining manufacturing test programs, and debugging tester programs.

2.5.4 Near Filter/Near Propagation

Near hazard analysis tests timing tolerances in the circuit, and attempts to determine whether minor variation in delays could cause a timing problem. This is essentially a "what-if" analysis that examines the effects of "close" transitions at element inputs (the definition of "closeness" is user-controllable). A description of near hazard analysis can be found in the Subsection *Combinational Timing Hazards* in Chapter 2.6. This is a robustness test, which may find problems that less critical methods of logic simulation miss. In near propagation mode, an X-pulse will be generated when a near hazard occurs. In near filter mode, no X-pulse will be generated. Near hazard propagation is controlled by the **`NEAR`** (**`NE`**) keyword option of the **`XPROPAGATE`** (**`XP`**) command. It is enabled with:

```
XPROPAGATE NEAR:
```

and disabled with:

```
NO XPROPAGATE NEAR:
```

The latter is SIMIC's default; if near propagation is not requested, SIMIC operates in near filter mode.

2.5.5 Spike Filter/Spike Propagation

A **spike hazard** occurs when an element output is scheduled to change and, before that time arrives, another event at the element's inputs causes the new output value to differ from the scheduled value. (This situation is also called a "glitch".) Since output delay time is primarily associated with a node's charging time, and since the occurrence of a spike indicates that the node had insufficient time to reach the previously-scheduled value, one method of handling spikes is to ignore the transient and maintain the node at its previous value. This model, called "inertial filtering", is utilized by many logic simulators. Spike filter mode, in effect, is equivalent to inertial filtering.

However, simulation is only a model of reality, and optimistic results may be obtained unless the effects of manufacturing tolerances are somehow incorporated. Spike propagation mode attempts to account for simulation uncertainties (possible differences between simulated delays and actual delays) by producing an X-pulse whenever a spike hazard occurs.

Spike propagation can be enabled or disabled globally with the **`SPIKE`** (**`SP`**) keyword option of the **`XPROPAGATE`** (**`XP`**) command. To disable spike propagation:

```
NO XPROPAGATE SPIKE:
```

and to enable it globally:

```
XPROPAGATE SPIKE:
```

By default, SIMIC propagates spikes. SIMIC actually allows the triggering condition and size of the X-pulse to be controlled on a per-node basis (see *Modifying Spike Control Parameters* in Chapter 2.6).

2.5.6 Stable After Decay/Stable Before Decay

This options banner entry reports the selected definition of **stability**. SIMIC performs certain operations when the simulated circuit's state becomes stable—it executes actions for which the **`PSTEP`** keyword option is active (**`PRINT`**, **`WRITE`**, and **`BREAK`** every *k*-th stable state) and applies the next input state for pattern mode stimuli.

By definition, the circuit state is stable when internal activity in response to the current input state has ceased. The question arises whether the long time-constants of charge decay should be included in this definition. The answer depends on the application.

Obviously, it would be impossible to simulate dynamic logic with stable-state patterns if the next pattern is applied only after all charged nodes are decayed. (However, if waveforms or timing generators are used with dynamic logic, it may be necessary to know whether a node does decay during the course of simulation.) For dynamic designs, the definition of stability should *not* include pending decays.

Conversely, a static design's lack of dependence on clock rate would not be fully verified *unless* all charge is decayed before applying the next pattern. For static designs, the definition of stability *should* include pending decays.

Thus, the application determines the definition of stability. The **DEFINE** (**DE**) command's **STABILITY** (**STAB**) keyword option is used for this purpose. To specify the that stability does not include pending decays, use the command:

```
DEFINE STABILITY=PREDECAY
```

To include pending decays in the definition of stability:

```
DEFINE STABILITY=POSTDECAY
```

The reserved word **PREDECAY** and **POSTDECAY** may be abbreviated to two letters.

By default, SIMIC includes pending decays in the definition of stability (**POSTDECAY**).

2.5.7 Dynamic Delays/Static Delays

Since the SIMIC ideal switch primitives (**BTGN** and **BTGP**) are resistanceless devices, nodes connected through ON ideal switches should behave as if they were physically connected. Therefore, by default, SIMIC dynamically sums the loading of nodes dynamically interconnected through ON ideal switches, and then recomputes the corresponding delays for the drivers of the tied nodes.

The **DEFINE** command's **BTGDELAY** (**BTGD**) keyword option controls dynamic delay computation. The following command disables dynamic delay modification:

```
DEFINE BTGDELAY=STATIC
```

To enable dynamic modification of driver delays:

```
DEFINE BTGDELAY=DYNAMIC
```

The reserved word **STATIC** and **DYNAMIC** may be abbreviated to one letter.

This option has no effect on simulation if there are no **BTGN** or **BTGP** elements or if the driver delays do not depend on loading.

Chapter 2.6 Circuit Troubleshooting

2.6.1 Introduction

SIMIC offers a wide selection of commands to facilitate circuit debugging. Each command performs a unique function that, when combined with other SIMIC options, creates a powerful debugging environment that can be customized to the problem at hand.

2.6.1.1 Debugging Capabilities

1. Detect specified conditions. This feature allows detection of a variety of events, such as signal change, presence of hazard conditions, reaching a specified test, and other situations. Upon the occurrence of the user-specified conditions, SIMIC can be instructed to issue a warning (using the **`WARN`** command), and/or stop simulation (using the **`BREAK`** command).
2. Interrogate the state of the network. This feature allows values (with the **`LOOK`** command) or parameters, such as delay (with the **`?`** command), to be requested for any signal, and forward (**`LOOK OUTPUT`**) or backward (**`LOOK INPUT`**) traversal through the circuit's topology.
3. Trace signal activity. Using the **`TRACE`** command, activity at selected signals can be monitored. Causality information (why a signal changed) can be requested (with the **`TRACE EXPAND`** command).
4. Enable/Disable hazard and/or timing checks on a per node basis.
5. Modify simulation parameters at run-time. This includes changing delay, decay, and node states (using the **`CLAMP`** and/or **`SET`** commands).
6. Simulate fragment. After a problem is localized, SIMIC allows restriction of actions to a small portion of the total simulation (with the **`PRANGE`** option) to prevent voluminous extraneous information. SIMIC also supports repetitive resimulation from a saved stable-state (**`RESTORE TNUM`** command), providing immediate feedback on modifications.

2.6.2 SIMIC Terminology and Definitions

2.6.2.1 Combinational Timing Hazards

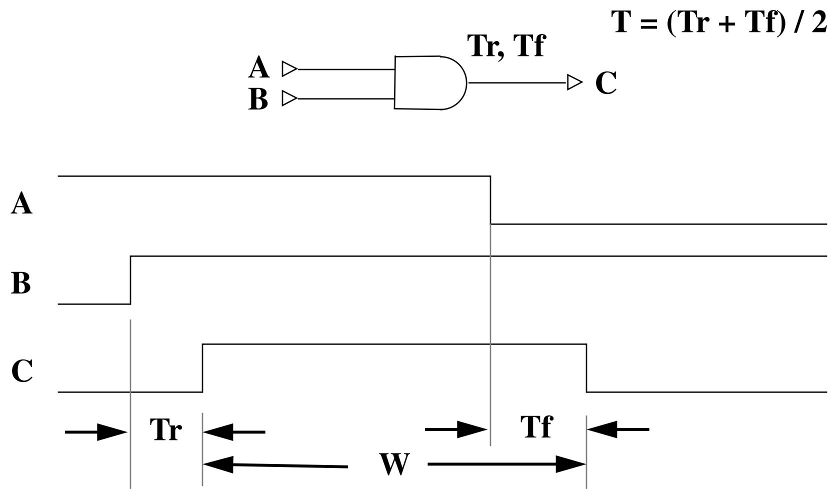
SIMIC performs several unique checks for timing hazards within the circuit. This section introduces the SIMIC terminology for these checks and describes their associated timing.

Figure 2.6-1 illustrates the supported combinational hazard checks for a two-input AND gate. In Figure 2.6-1(a), the output is a **pulse**, that is, a sequence of transitions starting from a known value (0 or 1), changing to the complementary value (or X), and then returning to the original value. If the pulse's width is comparable to the propagation delay of the gate during simulation, it may be narrower, or even nonexistent, in the physical circuit due to wiring delays, processing variations, and other factors. Narrow pulses are generally unplanned; if present, they could cause problems and therefore should be reported. By default, a **pulse hazard** is defined as a pulse that is no wider than three times the average propagation delay of the gate. This default can be changed to a different multiple, **<k>** (an integer), of the gate's average propagation delay with the **DEFINE** (**DE**) command's **PULSE** (**PU**) keyword option:

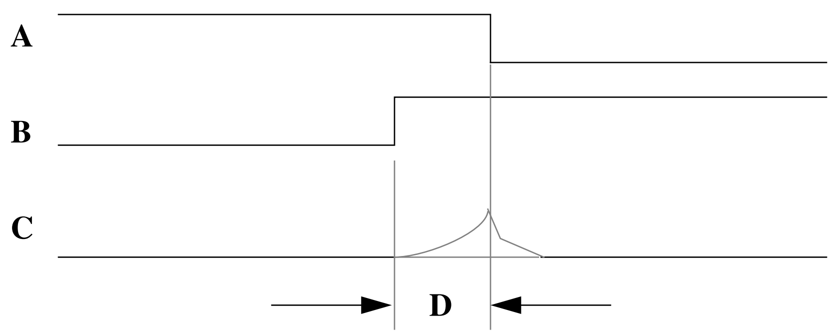
```
DEFINE PULSE=<k>
```

Figure 2.6-1(b) illustrates the situation where the input transitions are so close that the gate output cannot fully respond to the first input event before the second event occurs (a "pulse that never happened"). This is a called a **spike hazard**. Spike hazards at a memory elements can be specifically referenced with the **MEMSPIKE** keyword. SIMIC automatically counts the number of unreported spike conditions, and outputs this count at the end of each simulation run. By default, SIMIC generates an X-pulse at the node where the spike originates; this is called **spike propagation**. Alternatively, it can also be instructed to filter spike transients. Two spike propagation attributes within a SNL cell definition, **FILTER** and **LIBERAL**, regulate the degree of X-pulse pessimism on a per-node basis. These attributes are described in the Subsection *Controlling Spike Propagation* in Chapter 2.7. They can be modified interactively with the SIMIC **XPROPAGATE** run command.

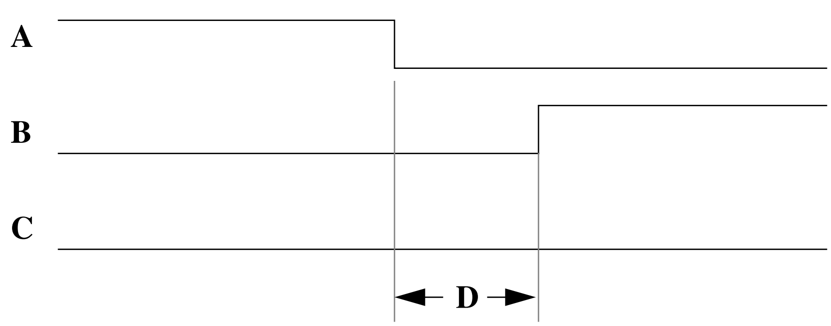
In Figure 2.6-1(c), the input transitions are in opposite order. Simulation results are "clean"; no activity is ever scheduled for the output. Yet, if these transitions are close, their arrival times may reverse order in the real circuit due to differences in arrival times, path delays, etc., from simulated values. Thus, events could also differ from simulated events. This is called a **near hazard**. SIMIC can be instructed to monitor near hazards and perform a "what if" analysis of input event ordering for combinational primitives whose inputs change within a predefined window. By default, this window is twice the average propagation delay of the gate.The window multiple,



(a) Pulse Hazard (0 ≤ W ≤ kT) where k = 3 (by default)



(b) Spike Hazard (0 < D < Tr) or (0 < D < Tf)



(c) Near Hazard (0 ≤ D ≤ kT) where k = 2 (by default)

Figure 2.6-1 Combinational Timing Hazards

<k> can be changed to a different value with the DEFINE command's **NEAR** (**NE**) keyword option:

```
DEFINE NEAR=<k>
```

SIMIC can optionally propagate an X-pulse upon detecting this hazard (see *Enabling And Disabling Spike Propagation* in this chapter).

2.6.2.2 Functional Timing Checks

Timing checks are supported for the DNL and DPL (DL) latch and the DNCF, DPCF (DCF), JKNCF (JKCF), JKPCF, TNCF (TCF), and TPCF edge-triggered flip-flops. These checks can be specified within a **TIMING–CHECKS** block in any PART statement instantiating these built-in primitives.

In the following description, the *active clock edge* is the clock transition that causes the latch or flip-flop output to change state. This edge is the rising clock transition for the DPL (DL), DPCF (DCF), JKPCF, and TPCF primitives, and the falling clock transition for the DNL, DNCF, JKNCF (JKCF), and TNCF (TCF) primitives.

The supported timing checks are:

1. **SETUP** – this check specifies the minimum duration that an input must be stable *prior to* an active clock edge:
   - DNL, DPL, DNCF, DPCF – setup from D (**SETUP.D**)
   - JKNCF, JKPCF – setup from J (**SETUP.J**) and setup from K (**SETUP.K**)
   - Additionally, all eight primitives support setup from reset (**SETUP.NR**), and setup from set (**SETUP.NS**). These setup times represent the minimum duration that the reset (set) must be *inactive* to reliably set (reset) the memory element via clock.
2. **HOLD** – this check specifies the minimum duration that an input must be stable *after* an active clock edge:
   - DNL, DPL, DNCF, DPCF – hold to D (**HOLD.D**)
   - JKNCF, JKPCF – hold to J (**HOLD.J**) and hold to K (**HOLD.K**)
   - Additionally, all eight primitives support hold to reset (**HOLD.NR**), and hold to set (**HOLD.NS**)
3. **PULSE–WIDTHS –** this check specifies the minimum width of a pulse on the set, reset, or clock lines. All eight primitives support: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

The setup and hold checks for the asynchronous, active-low, set and reset inputs of all eight primitives are associated with the time duration between the rising (trailing) edge of pulses on these inputs and the active clock edge.

Referencing a timing check by itself, without a qualifying pin name or clock level—**SETUP**, **HOLD**, **PW**—specifies *all* checks of that type. For example, in a **TIMING–CHECKS** block for a JKCF instance, **SETUP** specifies all setup checks; **SETUP.D**, **SETUP.J**, **SETUP.K**., **SETUP.NR**, and **SETUP.NS**.

2.6.2.3 Excessive Activity (Oscillation)

Oscillation is defined as excessive activity at a signal in response to a single change of primary input state. In order to determine excessive activity, SIMIC counts the state changes at each signal. When a primary input changes (or a new period starts for timing generators), all counts are reset. By default, oscillation is defined to be 10 transitions at a signal within a single test.

When activity at any signal reaches the defined oscillation limit, SIMIC sets the signal's value to X (unknown). The signal is freed to assume a known value when the next primary input change occurs.

This limit can be redefined globally with the **DEFINE** command's **OSCILLATION** (**OS**) keyword option:

    DEFINE OSCILLATION=<*count*>

where ***<count>*** is a number between 2 and 255. To disable oscillation detection, any prefix of the word **INFINITE** can be used instead of ***<count>***. This option should be exercised with extreme caution, since, if an oscillation does occur, the simulation run will never complete.

2.6.2.4 Depths and Strengths

SIMIC supports a range of 32,767 gradations of "resistance", called **depths**, for switch-level components. Depths are assigned to these components with the **SERIES–DEPTH** (**SDEPTH**) SNL keyword. Smaller values of depth ("resistance") correspond to greater drive capability.

Since gate-level components can interconnect with switch-level components, it is necessary to establish a correspondence between switch-level depths and gate-level drive strengths. Depths are mapped to strengths as follows:

Depth Magnitude	Strength
1	Power
2	Driving
3-32,766	Resistive
32,767	Floating

When SIMIC needs to map a strength to a depth value, Resistive strength is mapped to a depth of 3, by default. The default value can be changed with the DEFINE command's **RDEPTHS** (**RD**)keyword option:

```
DEFINE RDEPTH=<n>
```

where ***<n>*** is a value between 3 and 32766.

2.6.2.5 Interval Representation

In order to perform switch-level simulation accurately, SIMIC internally maintains an interval representation for switch-level signal values. The interval contains information on the range of possible values at each signal.

Each interval is represented by a two-tuple of integers whose absolute values range from 1 to 32,767. The integers contain combined depth and level information. If both integers are positive, the level is logical-1. If both integers are negative, the level is a logical-0. Otherwise, the level is an X. The absolute magnitudes of the integers represent depths—the lower the value, the stronger the drive.

Some examples of intervals:

- **2/2** - this represents a driving 1.
- **-3/-3** - this represents a resistive 0.
- **2/-2** - this represents a driving X.
- **2/20** - this represents a logical-1, whose strength is somewhere between 2 and 20.
- **-2/-20** - this represents a logical-0, whose strength is somewhere between 2 and 20.
- **32767/-32767** - this represents a tristate (Z) condition.

An interval of **2/20**, for example, can arise when a node is driven to logical-1 through a depth of 20, and *may* also be driven to logical-1 through a depth of 2. One situation that can generate this interval is illustrated in Figure 2.6.2.

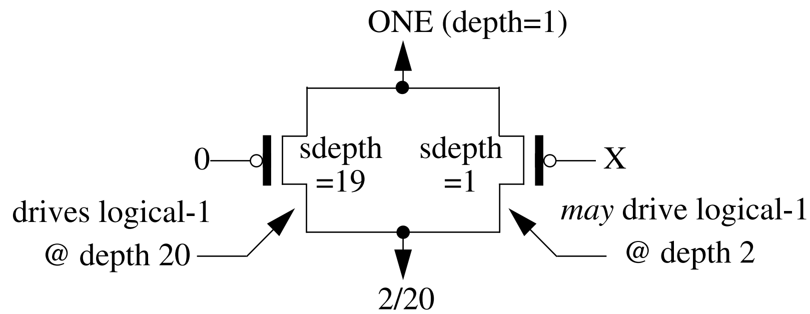


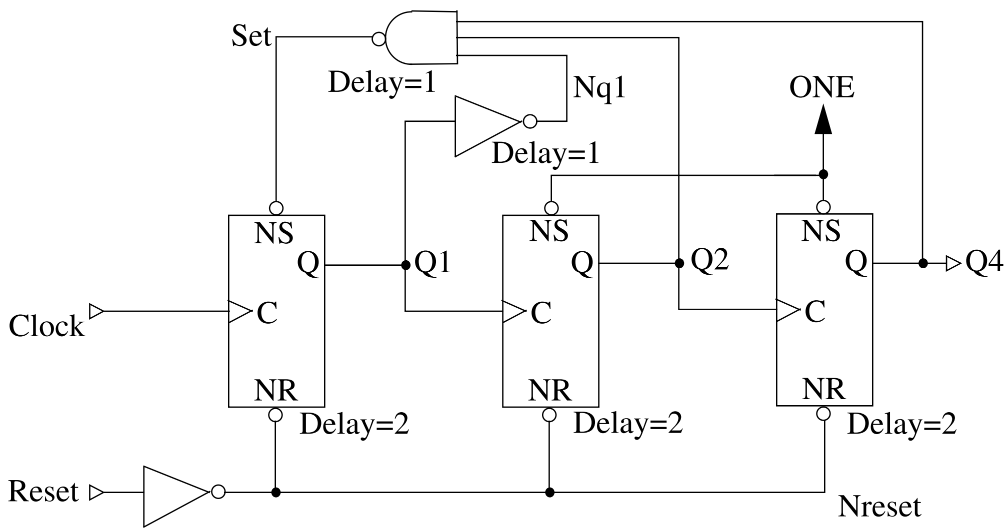
Figure 2.6-2 Sample Situation Generating 2/20 Interval

For intervals with unequal components (e.g., 2/20, representing a range of possible depths), SIMIC uses the following simulation output characters:

- F - for logical-0.
- T - for logical-1.
- U - for X (unknown).

2.6.3 Interactive Debugging Example

The Divide-by-7 counter shown in Figure 2.6-3 has a potential timing hazard that only becomes apparent when spikes are allowed to propagate. This indicates a timing problem that may manifest itself in non-functioning devices, or in reduced yields. This example demonstrates the necessity of a robust spike propagation algorithm. In addition, it demonstrates some of SIMIC's interactive debugging capabilities, and provides a brief description of these capabilities. The remainder of this chapter contains detailed descriptions of these debugging features, and others.




```
C= SNL Description of divide by 7 circuit
!f p=    t=   i=              o= ochange=
type=divide_by_7 i=reset,clock o=q1,q2,q4
  nreset inv  reset           -  1
  q1     tcf  nreset,set,clock q1 2
  q2     tcf  nreset,one,q1    q2 2
  q4     tcf  nreset,one,q2    q4 2
  nq1    inv  q1              -  1
  set    nand nq1,q2,q4       -  1
```

Figure 2.6-3 Divide-by-7 Counter

In this circuit, three negative edge T flip-flops (built-in primitive TCF, see Appendix A) are initially reset to a count of 000 (**q4**,**q2**,**q1**). When the counter reaches state 110 on the sixth negative-going **clock** edge, signal **set** goes low, forcing a counter state of 111, which makes signal **set** go high again. The next **CLOCK** pulse causes the counter state to return to 000. (Note: the **OCHANGE**, or **OUTPUT-CHANGE** SNL keyword assigns identical rise and fall delays to an output; thus, signal **nreset** has a delay of 1, signal **q1** has a delay of 2, etc. See *Local Delays* in Chapter 2.7)

```
>>: define file=div
>>: get type=divide_by_7

Main Get Network : DIVIDE_BY_7
GET completed, Circuit totals: Parts = 6; Signals = 10
   Inputs = 2; Busses = 0; Outputs = 3
>>: define pr.1 = 1 0
>>: define pc.1 = 0 0 do 8 (1 0)
>>: apply patterns=pc list=clock
>>: apply patterns=pr list=reset
>>: print li=reset*clock**q1,q2,q4*set
>>: no xpropagate spike:
>>: simulate

Remark= Options: (Fault Free Simulation)
Remark= Pattern Stimuli, Near Filter, Spike Filter
Remark=   Stable After Decay, Dynamic Delay

C=                R C  QQQ S
C=                E L  124 E
C=                S O      T
C=                E C
C=                T K

        7 T     1: 1 0  000 1
        1 T     2: 0 0  000 1
        2 T     3: 0 1  000 1
        4 T     4: 0 0  100 1
        2 T     5: 0 1  100 1
        6 T     6: 0 0  010 1
        2 T     7: 0 1  010 1
        4 T     8: 0 0  110 1
        2 T     9: 0 1  110 1
        6 T    10: 0 0  001 1
        2 T    11: 0 1  001 1
        4 T    12: 0 0  101 1
        2 T    13: 0 1  101 1
        9 T    14: 0 0  111 1
        2 T    15: 0 1  111 1
        6 T    16: 0 0  000 1
        2 T    17: 0 1  000 1
        4 T    18: 0 0  100 1
2 Spike messages suppressed.
```

Figure 2.6-4 Divide-by-7 Counter — Run 1

By default, SIMIC propagates an X-pulse whenever a spike hazard occurs. To demonstrate the importance of X-pulse propagation, the first simulation of the Divide-by-7 circuit is performed *without* it. This is accomplished by issuing the **`NO XPROPAGATE SPIKE:`** run command (spike filter mode).

This simulation, called Run 1, is shown in Figure 2.6-4. Based on these results, which would be obtained from simulators that do not support spike propagation, the counter appears to behave as predicted, and therefore the design might be assumed to be correct and problem-free.

Although spike propagation was inhibited, SIMIC still issued a warning that two unreported spikes occurred during simulation. In general, SIMIC will count the number of unreported spikes that occur during simulation (even if spike warning messages have been suppressed or have not been requested) and will report this number at the end of each simulation run.

Because of the post-simulation warning that unreported spikes occurred, the simulation is repeated, this time allowing spikes to propagate. This is accomplished simply by eliminating the **`NO XPROPAGATE SPIKE:`** command, since spike propagation is SIMIC's default.

This simulation session, called Run 2, is shown in Figure 2.6-5. As a result of spike propagation, **`q1`**, **`q2`**, and **`q4`** went unknown (X) at test 16. This is an indication that the spikes do indeed introduce a critical timing problem.

Having determined that a problem exists, the next step is to perform interactive debugging to determine its cause. The simulation will be run again, and a breakpoint will be set to "freeze" the simulation when a spike occurs. This is accomplished with the

```
BREAK SPIKE: MEMSPIKE:
```

run command. SIMIC can distinguish spikes in memory elements (**`MEMSPIKE`**) from other spikes (**`SPIKE`**). The **`SPIKE:`** option in the **`BREAK`** command requests a break from simulation if *any* spike (including spikes in memory elements) is occurs. Although this would have been sufficient for this example, the **`MEMSPIKE:`** option is used to request a break if a spike occurs in a flip-flop primitive (a subset of the **`SPIKE:`** option). As a result, a spike at a memory element will be reported as a **`MEMSPIKE`**, rather than as a **`SPIKE`**.

```
>>: define file=div
>>: get type=divide_by_7

Main Get Network : DIVIDE_BY_7
GET completed, Circuit totals: Parts = 6; Signals = 10
   Inputs = 2; Busses = 0; Outputs = 3
>>: define pr.1 = 1 0
>>: define pc.1 = 0 0 do 8 (1 0)
>>: apply patterns=pc list=clock
>>: apply patterns=pr list=reset
>>: print li=reset*clock**q1,q2,q4*set
>>: simulate

Remark= Options: (Fault Free Simulation)
Remark=   Pattern Stimuli, Near Filter, Spike Propagation
Remark=   Stable After Decay, Dynamic Delay

C=                 R C  QQQ S
C=                 E L  124 E
C=                 S O      T
C=                 E C
C=                 T K

        7 T     1: 1 0  000 1
        1 T     2: 0 0  000 1
        2 T     3: 0 1  000 1
        4 T     4: 0 0  100 1
        2 T     5: 0 1  100 1
        6 T     6: 0 0  010 1
        2 T     7: 0 1  010 1
        4 T     8: 0 0  110 1
        2 T     9: 0 1  110 1
        6 T    10: 0 0  001 1
        2 T    11: 0 1  001 1
        4 T    12: 0 0  101 1
        2 T    13: 0 1  101 1
        9 T    14: 0 0  111 1
        2 T    15: 0 1  111 1
       10 T    16: 0 0  XXX X
        0 T    17: 0 1  XXX X
        0 T    18: 0 0  XXX X
2 Spike messages suppressed.
```

Figure 2.6-5 Divide-by-7 Counter — Run 2

Figure 2.6-6 illustrates the results of this simulation, called Run 3. SIMIC stops the simulation at test 16, issues two **MEMSPIKE BREAK** messages, and reports that spikes originate at both the master, **q1.1**, and slave, **q1**, of flip-flop **q1**.

At this point, the simulation is frozen at time 5 of test 16, and SIMIC has issued its prompt requesting a command. In order to obtain more information about the cause of the spike, the circuit's response to test 16 will be resimulated.This time, however, all activity will be traced to determine the sequence of events that lead to the spike. As before, SIMIC will break from simulation when the spike occurs, since the **BREAK** command is still active (the **BREAK** can only be removed with a **NO BREAK** command).

The command:

```
restore tnum=*
```

rolls time back to the beginning of the current test (the last stable state), so when the **SIMULATE** command is issued, test 16 will be resimulated.

The command:

```
trace list: expand:
```

causes the activity at every signal to be traced. Additionally, use of the **EXPAND** option causes the trace output to be augmented with causality information.

```
>>: define file=div
>>: get type=divide_by_7

Main Get Network : DIVIDE_BY_7
GET completed, Circuit totals: Parts = 6; Signals = 10
   Inputs = 2; Busses = 0; Outputs = 3
>>: define pr.1 = 1 0
>>: define pc.1 = 0 0 do 8 (1 0)
>>: apply patterns=pc list=clock
>>: apply patterns=pr list=reset
>>: print li=reset*clock**q1,q2,q4*set
>>: break spike: memspike:
>>: simulate

Remark= Options: (Fault Free Simulation)
Remark=  Pattern Stimuli, Near Filter, Spike Propagation
Remark=   Stable After Decay, Dynamic Delay

C=                 R C  QQQ S
C=                 E L  124 E
C=                 S O      T
C=                 E C
C=                 T K

        7 T     1: 1 0  000 1
        1 T     2: 0 0  000 1
        2 T     3: 0 1  000 1
        4 T     4: 0 0  100 1
        2 T     5: 0 1  100 1
        6 T     6: 0 0  010 1
        2 T     7: 0 1  010 1
        4 T     8: 0 0  110 1
        2 T     9: 0 1  110 1
        6 T    10: 0 0  001 1
        2 T    11: 0 1  001 1
        4 T    12: 0 0  101 1
        2 T    13: 0 1  101 1
        9 T    14: 0 0  111 1
        2 T    15: 0 1  111 1
        5 B    16> MEMSPIKE(0->1->0) Q1.1
        5 B    16> MEMSPIKE(0->1->0) Q1
>>:
```

Figure 2.6-6 Divide-by-7 Counter — Run 3

```
>>: restore tnum=*
>>: trace list: expand:
>>: simulate

Remark= Options: (Fault Free Simulation)
Remark=   Pattern Stimuli, Near Filter, Spike Propagation
Remark=   Stable After Decay, Dynamic Delay

C=                R C  QQQ S
C=                E L  124 E
C=                S O      T
C=                E C
C=                T K

        0 E    16> TRACE(1->0) CLOCK
        2 E    16> TRACE(1->0) Q1
          C      >      (1->0) CLOCK
        3 E    16> TRACE(0->1) NQ1
          C      >      (1->0) Q1
        4 E    16> TRACE(1->0) Q2
          C      >      (1->0) Q1
        4 E    16> TRACE(1->0) SET  <-- beginning of pulse at SET
          C      >      (0->1) NQ1  <--   causality
          C      >      (1->0) Q2   <--   causality
        5 E    16> TRACE(0->1) SET  <-- end of pulse at SET
          C      >      (1->0) Q2   <--   causality
        5 B    16> MEMSPIKE(0->1->0) Q1.1
        5 B    16> MEMSPIKE(0->1->0) Q1
        5 E    16> TRACE(0->X) Q1.1
          C      >      (0->1) SET
        5 E    16> TRACE(0->X) Q1
          C      >      (0->1) SET
>>:
```

Figure 2.6-7 Divide-by-7 Counter — Run 3 (continued)

The results of resimulating test 16 with activity trace are shown in Figure 2.6-7. (Note that this is still the same session, Run 3.) SIMIC breaks from simulation at time 5 of test 16, as before. Even a cursory examination of the **`TRACE`** output indicates that a unit-width pulse occurs on signal **`set`**; it executes a 1→0 transition at time 4 and a 0→1 transition at time 5. This pulse is not wide enough to actually set flip-flop **`q1`**, whose rise delay is 2 time-units.

Causality information, in turn, indicates that the pulse (which shouldn't exist at all) is due to a race condition between **`nq1`** and **`q2`**. Therefore, it should be possible to eliminate the spike by increasing the delay of **nq1** to make its 0→1 transition occur *after* the transition at **q2**. The required delay modification would be from its current value of 1 to any value greater than 2 (say, 3).

At any point when the circuit is in a "frozen" state (simulation is stopped by a break), commands can be entered to modify parameters and/or signal values as well as to request information about the circuit's state, topology and electrical characteristics.

Figure 2.6-8 shows the simulation results when this fix is made interactively. As before, the command:

```
restore tnum=*
```

rolls time back to the beginning of the current test, so when the **SIMULATE** command is issued, test 16 will be resimulated. The propagation delay of **NQ1** is modified with the command:

```
set change=3 list=nq1
```

The activity trace, no longer necessary, is removed with the command:

```
no trace list:
```

Resimulation shows that, indeed, no spike occurs with this modification:

```
>>: restore tnum=*
>>: set change=3 list=nq1
>>: notrace list:
>>: simulate

Remark= Options: (Fault Free Simulation)
Remark=   Pattern Stimuli, Near Filter, Spike Propagation
Remark=   Stable After Decay, Dynamic Delay

C=                  R C  QQQ S
C=                  E L  124 E
C=                  S O      T
C=                  E C
C=                  T K


        6 T    16: 0 0  000 1
        2 T    17: 0 1  000 1
        5 T    18: 0 0  100 1
>>:
```

Figure 2.6-8 Divide-by-7 Counter — Run 3 (continued)

At this point, it is suggested that the simulation be repeated for the full stimulus sequence, to make sure that no new problems have been introduced at other points in the test sequence.

2.6.4 Restricting Simulation Options To A Specified Simulation Interval

2.6.4.1 Commands Affected

SIMIC has a general mechanism to restrict many of its options to a given test range (for patterns) or time interval (for waveforms). This is done with the **`PRANGE`** (**`PR`**) keyword option in the individual commands. The commands that can utilize this option are:

- BREAK
- HISTORY
- PRINT
- SAVE
- SIMULATE
- TRACE
- WARN
- WRITE

`PRANGE` specifications for these commands are independent; restricting the active interval of one command does not affect the active interval of the other commands.

2.6.4.2 Basic Form of PRANGE Keyword

The **`PRANGE`** keyword option accepts either integers or intervals of integers. These numbers represent test numbers for pattern stimuli and simulation times for waveform stimuli.

A **`PRANGE`** interval is specified as:

<starting point>−<ending point>

For example, to restrict output to the **write** file from test (time) 100 to test (time) 200 only:

```
WRITE PRANGE=100-200
```

If the option is to be restricted to one test (time) then just the ***<starting point>*** need be entered. For example:

```
WRITE PRANGE=1000
```

is equivalent to:

```
WRITE PRANGE=1000-1000
```

and would only enable **write** output at test (time) 1000. Multiple intervals can be specified:

```
WRITE PRANGE=10-20,200-220,300-310,500
```

PRANGE options are “sticky”, meaning a previously specified interval must be explicitly “undone” with the **NO** prefix, and **PRANGE** specifications are cumulative. In the above example, the command could be broken into an equivalent four command sequence:

```
WRITE PRANGE=10-20
WRITE PRANGE=200-220
WRITE PRANGE=300-310
WRITE PRANGE=500
```

When a **NO** prefix is used with a command, it excludes the **PRANGE** interval. For instance, the command:

```
WRITE PRANGE=1000-2000
```

would enable **write** output from test (time) 1000 to test (time) 2000. Subsequently, the command:

```
NO WRITE PRANGE=1101-1199
```

would disable the **write** output from test (time) 1101 to test (time) 1199. The two commands are equivalent to the single command:

```
WRITE PRANGE=1000-1100,1200-2000
```

The colon form of the **PRANGE** keyword:

```
PRANGE:
```

specifies *all* tests (time).

If no **PRANGE** option is specified for a command, then SIMIC defaults to “enabled for the entire simulation range” (equivalent to **PRANGE:**).When the **PRANGE** option is used the first time for a given command, and the **NO** command prefix is not specified, then the command’s active range will be the specified **PRANGE** interval. From then on, **PRANGE** options for that command will be cumulative (merged with the previous specifications).

If ***<starting point>*** is omitted, 0 is assumed. SIMIC interprets a 0 entry to mean “at the pre-simulation point”, even for patterns. If ***<ending point>*** is omitted, then the interval is open-ended (until the last input stimulus state has been propagated). However, either ***<starting point>*** or ***<ending point>*** must be specified. Open-ended and 0-starting intervals are specified with a hyphen (-). For example:

```
NO WRITE PRANGE=1000-
```

specifies that the **write** output will be suppressed from test (time) 1000 until the end of simulation and

```
SIM PRANGE=-20000
```

specifies that SIMIC should stop applying new input states after test (time) 20,000. Note: after the last stimulus in this range has been applied, simulation will still continue until all internal circuit activity ceases.

2.6.5 Setting Simulation Breakpoints

2.6.5.1 Overview

One of the most basic debugging facilities in SIMIC is the ability to "freeze" the simulation at a specified point for interrogation. When any of the specified conditions for "freezing" simulation have been met, SIMIC issues a break message in the following form:

<*time*> B <*test*> > <*description*>

where ***<test>*** is the test and ***<time>*** is the time when the condition occurred. The **B** delimiter specifies that this is a **BREAK** message. The '**>**' delimits the **<time>** and **<description>** fields, where **<description>** contains the condition that caused the break (**SPIKE**, **CONFLICT, RISE**, etc.) and the part or signal that generated the condition. For example, the **BREAK** message in the above divide-by-7 simulation:

```
5 B    16> MEMSPIKE(0->1->0) Q1.1
```

indicates a **MEMSPIKE** condition (spike hazard at a memory element) on signal **Q1.1** at time 5 of test 16, and the polarity of the spike (0→1→0).

2.6.5.2 Restricting Break To A Specified Interval

BREAK operation can be restricted to a specified test (patterns) or time (waveforms) with the **PRANGE** option of the **BREAK** command. See the Section *Restricting Simulation Options To A Specified Simulation Interval* covering **PRANGE** specifications earlier in this chapter.

2.6.5.3 Directing The Destination Of Break Messages

By default, **BREAK** messages are displayed at the terminal. In addition, these messages may be sent to a file whose default extension is **brk**. This is accomplished with the **FILE** (**FI**) keyword option of the **BREAK** (**BR**) command. The file's name can be obtained from the default file name with:

```
BREAK FILE:
```

or can be explicitly specified with:

BREAK FILE=<*name of file*>

where ***<name of file>*** is the file's explicit name.

Output to the file can be disabled with the command:

```
NO BREAK FILE:
```

The **FILE** option does not affect messages directed to the terminal. Terminal output is controlled with the **TERM** (**TE**) keyword option. This output can be disabled with the command:

```
NO BREAK TERM:
```

The following command restores terminal output:

```
BREAK TERM:
```

2.6.5.4 Breakpoint At A Specified Signal Transition

A breakpoint can be set when a signal executes a rise (0→1, X→1, or 0→X) transition with the **RISE** (**RI**) keyword option or a fall (1→0, X→0, or 1→X) transition with the **FALL** (**FA**) keyword option. Additionally, any change in value can be trapped with the **CHANGE** (**CH**) keyword. The format is:

BREAK <*keyword*>=<*list of signals*>

where ***<keyword>*** is either **RISE**, **FALL**, or **CHANGE**, and ***<list of signals>*** is a list of signals for which the option is applied (or removed with the **NO** prefix to the **BREAK** command). Using:

<*keyword*>**:**

adds (removes) the specified option for all signals.

To disable a breakpoint on any of these transitions, use the **NO** prefix to the **BREAK** command and the appropriate keyword and values.

2.6.5.5 Breakpoint When A Signal State Becomes Unknown (X)

A breakpoint can be set when specified signals go to X. This is accomplished with the **X** keyword, using the same format as the above keywords:

BREAK X=<*list of signals*> or BREAK X:

In addition to the normal breakpoint information, causality information will be included in the message.

The **MEMLATCH** (**MEML**) keyword can be used to set breakpoints when known-to-unknown state transitions occur at SIMIC latch and flip-flop primitives as a result of sensitized unknown inputs (e.g., X at the clock or asynchronous set/reset inputs). Other keywords can be used to set breakpoints when the transition to an unknown state is due to timing problems at the memory element's inputs (see *Breakpoint At A Combinational Timing Hazard* and *Breakpoint At A Functional Timing Violation* below). The format is:

BREAK MEMLATCH=<*list of signals*>

where ***<list of signals>*** is a list of SIMIC latch and flip-flop outputs. To set a breakpoint when the state of *any* memory element becomes unknown due to a sensitized unknown input, use the command:

```
BREAK MEMLATCH:
```

To disable a breakpoint on any of these transitions, use the **NO** prefix to the **BREAK** command and the appropriate keyword and values.

2.6.5.6 Breakpoint When A Signal Goes To Floating Unknown (Z)

Sometimes it is helpful to monitor nodes that should never decay to a Z value (e.g., signals that drive unidirectional combinational gates). SIMIC can trap for this condition with the **DECAY** (**DE**) keyword option for the **BREAK** run command. The format is:

`BREAK DECAY=`<*list of signals*>

where ***<list of signals>*** specifies the signals to monitor, or

`BREAK DECAY:`

to monitor all signals. Subsequently, a monitor may be removed by prefixing the **BREAK DECAY** run command with the **NO** prefix, and the appropriate keyword value (either colon (**:**), or equals sign (**=**) followed by ***<list of signals>***).

2.6.5.7 Breakpoint At Signal Conflict Hazard

A conflict hazard occurs when two or more of the current highest strength drivers of a wire-tie are at complementary (or possibly complementary) values. To set a breakpoint for this situation, use the **CONFLICT** (**CON**) keyword option of the **BREAK** command. Individual signals may be selected for triggering a break with the

`BREAK CONFLICT=`<*list of signals*>

option, and all signals can be selected with the

`BREAK CONFLICT:`

option. Prefixing the **BREAK** command with a **NO** prefix will cause de-selection, rather than selection.

2.6.5.8 Breakpoint At An Oscillation

To enable a breakpoint when any signal oscillates (exhibits excessive behavior), use the **OSCILLATION** (**OS**) keyword option of the **BREAK** command:

`BREAK OSCILLATION:`

To disable this break:

`NO BREAK OSCILLATION:`

2.6.5.9 Breakpoint At A Combinational Timing Hazard

A breakpoint can be set for the timing hazards **PULSE** (**PU**), **SPIKE** (**SP**), and **NEAR** (**NE**), (described in the Subsection *Combinational Timing Hazards* in this chapter) by the run command:

`BREAK` <*keyword*>= <*list of signals*>

where ***<keyword>*** is either **PULSE**, **SPIKE**, or **NEAR** respectively, and ***<list of signals>*** are the signals for which a breakpoint is set. The format:

`BREAK` <*keyword*>**:**

specifies that the ***<keyword>*** hazard breakpoint is to be applied to all signals.

In addition, SIMIC provides two special keywords **HAZARD** (**HA**) and **MEMSPIKE** (**MEMS**). The **HAZARD** keyword is effectively equivalent to issuing the **SPIKE**, **PULSE** and **NEAR** keywords concurrently, with the same values. Thus:

```
BREAK HAZARD:
```

is equivalent to:

```
BREAK SPIKE: PULSE: NEAR:
```

The **MEMSPIKE** keyword is effectively equivalent to the **SPIKE** keyword but applies only to flip-flops. In addition, if this keyword is used, the specified flip-flops will generate a **MEMSPIKE** message rather than a **SPIKE** message.

To disable a breakpoint on any of these hazards, use the **NO** prefix to the **BREAK** command and the appropriate keyword and values.

2.6.5.10 Breakpoint At A Functional Timing Violation

Breakpoints can be set for setup, hold and pulse-width timing check violations with the **PART** (**PA**) keyword option and the appropriate timing check name. The run command:

BREAK PART: *<timing-check>*:

sets breakpoints for the specified ***<timing-check>*** violation for all parts, while

BREAK PART=*<list of parts>* *<timing-check>*:

sets breakpoints for the specified ***<timing-check>*** violation for the designated ***<list of parts>.***

<timing-check> is a supported timing check (setup, hold or pulse-width), either qualified or unqualified. The complete timing check name must be used. For example, to set breakpoints for violations of the D-setup-time check and all pulse-width checks for the DCF instance named **f1**, use the command:

```
BREAK PART=f1 SETUP.D: PW:
```

To disable a previously-enabled breakpoint on any timing violation, use the **NO** prefix to the **BREAK** command and the appropriate keyword and values.

See Appendix A for information on the functional timing checks supported for each primitive.

2.6.5.11 Breakpoint On Input Change While The Circuit Is Unstable

The **UNSTABLE** (**UN**) keyword option may be used to set a breakpoint when a primary input changes while the circuit is still unstable:

```
BREAK UNSTABLE:
```

To disable this breakpoint:

```
NO BREAK UNSTABLE:
```

2.6.5.12 Breakpoint At Specified Intervals

SIMIC always notes the event that the circuit state has become stable. SIMIC can be directed to break periodically at any of these stable points by using the **PSTEP** (**PS**) keyword option of the **BREAK** command. The form is:

```
BREAK PSTEP=<n>
```

where ***<n>*** is the number of stable points between each breakpoint.

When using waveforms, the circuit might not reach stability at the point of interest. In this case, combining the **PSTEP** and the **PRANGE** options could be useful. For example, if detailed analysis is required around time 1,000, the command:

```
BREAK PSTEP=1 PRANGE=900-
```

will stop the simulation the first time the circuit becomes stable after time 899.

To disable a **PSTEP** break use the command:

```
NO BREAK PSTEP:
```

Periodic breakpoints can also be set at a specified *time* interval with the **TSTEP** (**TS**) keyword option of the **BREAK** command. The form for this command is:

```
BREAK TSTEP=<n>
```

where ***<n>*** is the interval between breaks. For example, to stop the simulation every 10 time-units, use the command:

```
BREAK TSTEP=10
```

To disable an active **TSTEP** breakpoint, use the command:

```
NO BREAK TSTEP:
```

2.6.5.13 Breakpoint On Strobe Error

In Timing generator mode, SIMIC emulates the tester environment, including edge strobes (strobe point) and window strobes (strobe window). The strobed value is expected to remain constant (stable) during the entire active strobe interval. If a strobed signal does change value within this interval, then SIMIC can trap this event as a **strobe error**.

The **STROBE** (**STRO**) keyword option of the **BREAK** command may be used to set a breakpoint when a strobe error occurs:

```
BREAK STROBE:
```

To remove the breakpoint, use:

```
NO BREAK STROBE:
```

A full discussion of strobes, along with other topics related to test equipment interface, can be found in Chapter 2.8 of this Guide.

2.6.6 Setting Simulation Warnings (Watchpoints)

2.6.6.1 Overview

For many situations, it is sufficient to receive warning message and continue simulation, rather than to break from simulation. The SIMIC **WARN** (**WA**) command provides the ability to monitor user-specified simulation conditions and issue warnings if and when these conditions occur. Syntactically, the **WARN** command is similar to the **BREAK** command. Even the format of the warning messages:

<*time*> `W` <*test*> `>` <*description*>

is identical, with the exception that **W** is used instead of **B** as the delimiter between ***<time>*** and ***<test>***.

The default extension for the **WARN** file is **wrn**.

The majority of the options supported for the **BREAK** command are also supported for the **WARN** command. The options that are *not* supported are:

1. Warnings at a specified signal transition (**RISE**, **FALL**, **CHANGE**, **X**).
2. Warning at specified intervals (**PSTEP**, **TSTEP**).

2.6.6.2 Suppressing Excessive Messages On A Per-Signal Basis

SIMIC maintains a counter for most warnings on a per-signal basis. Once the number of warning messages for a specified condition have reached a limit (default 10) for a signal, SIMIC suppresses further messages of that type. For example, SIMIC will only display 10 SPIKE (spike hazard) messages, 10 PULSE (pulse hazard) messages, etc., for each signal. A new global limit for per-signal messages can be set with the WARN command's **STOP** (**STOP**) keyword option:

`WARN STOP=`<*n*>

where ***<n>*** is the new limit, ranging from 1 to 511. To prevent any messages from being suppressed, use the command:

```
NO WARN STOP:
```

For the special case of SPIKEs, the number of warning messages suppressed because a per-signal limit has been reached, or because spike warnings were not requested for a signal, is displayed at the end of each simulation run.

2.6.6.3 Warning Defaults

Warning messages for oscillations (**OSCILLATION**), wire-tie conflicts (**CONFLICT**), strobe errors (**STROBE**), primary input changes while the circuit is unstable (**UNSTABLE**), and all part timing checks (**SETUP**, **HOLD**, **PW**) are enabled by default. All other warnings are initially disabled.

2.6.7 Tracing Circuit Activity

2.6.7.1 Overview

Once the test has been found in which an incorrect or unexpected event occurs, the ability to follow the circuit's activity all the way to the original cause, with the **TRACE** (**TR**) command, is extremely useful. Each line of **TRACE** output contains the time, the test, and a description of the event at a traced signal. If requested, the cause (or probable causes) of the event can also be reported. The format of the **TRACE** output is:

```
<time> E <test> > TRACE (<transition>) <signal>
```

where ***<test>*** is the current test, ***<time>*** is the current time, ***<transition>*** contains the previous and new values describing the event, and ***<signal>*** is the traced signal's name.

If causality information is also requested (with the **TRACE EXPAND:** command), additional trace output will have the form:

```
C        > (<transition>) <signal>
```

For example, in the Divide-by-7 debugging session illustrated earlier in this chapter, the **TRACE** output:

```
2 E    16> TRACE(1->0) Q1
  C      >      (1->0) CLOCK
```

indicates: at time 2 in test 16, **Q1** executed a $1 \rightarrow 0$ transition, caused by **CLOCK** executing a $1 \rightarrow 0$ transition.

2.6.7.2 Restricting Trace To A Specified Interval

TRACE operation can be restricted to a specified test (patterns) or time (waveforms) interval with the **PRANGE** option of the **TRACE** command. See the Section *Restricting Simulation Options To A Specified Simulation Interval* covering **PRANGE** specifications earlier in this chapter.

2.6.7.3 Directing The Destination Of Trace Output

By default, **`TRACE`** messages are displayed at the terminal. In addition, this output may be sent to a file whose default extension is **trc**. This is accomplished with the **`FILE`** (**`FI`**) keyword option of the **`TRACE`** command. The file's name can be obtained from the default file name with:

```
TRACE FILE:
```

or can be explicitly specified with:

`TRACE FILE=`<*name of file*>

where ***<name of file>*** is the file's name.

The **`FILE`** option does not affect messages directed to the terminal.

`TRACE` output to the file can be stopped with the command:

```
NO TRACE FILE:
```

`TRACE` output to the terminal is controlled with the **`TERM`** (**`TE`**) keyword option. This output can be stopped with the command:

```
NO TRACE TERM:
```

and resumed with the command:

```
TRACE TERM:
```

2.6.7.4 Specifying Signals To Trace

The **`LIST`** (**`LI`**) keyword option selects the signals to be traced. To specify that all signals should be traced, simply issue the command:

```
TRACE LIST:
```

To select individual signals for tracing, use the command:

`TRACE LIST=`<*list of signals*>

where ***<list of signals>*** is the list of the signals to trace.

Similarly, to selectively stop signals from being traced, use the command:

`NO TRACE LIST=`<*list of signals*>

To stop tracing on all signals, use the command:

```
NO TRACE LIST:
```

2.6.7.5 Requesting Causality Information

Causality information is a useful backtracing facility, allowing traversal of the trace forwards and backwards. The **`EXPAND`** (**`EX`**) keyword option controls reporting of causality. To include causality information in the trace output, use the command:

```
TRACE EXPAND:
```

To remove causality information from the trace output, use the command:

```
NO TRACE EXPAND:
```

2.6.7.6 Using Trace To Locate Critical Paths

This feature is only applicable to the simulate-till-stable (patterns) simulation mode. If, in a given test, the circuit is expected to reach a stable state within some time limit, but its response time is found to exceed this limit, the critical path can be isolated by enabling activity trace only after the expected response time.

For example, if the circuit was expected to take 1000 time-unit to stabilize, but SIMIC simulation reveals that the circuit requires 1090 time-units to stabilize, then all activity after time 1000 can be traced (and all activity prior to this time not traced) by using the **BEGIN** (**BE**) keyword option of the **TRACE** command. In this case the command would be:

```
TRACE LIST: BEGIN=1001
```

Thus any signal that changed after time 1000 would generate a trace output, indicating the slowest paths in the circuit.

To disable the **BEGIN** option, issue the command:

```
NO TRACE BEGIN:
```

2.6.8 Probing For Signal State Information

2.6.8.1 Overview

The value of any signal in the circuit may be interrogated with the **LOOK** (**LO**) run command at any breakpoint in the simulation. The **LIST** (**LI**) keyword option is used to specify the signals. The format of this command is:

LOOK LIST=<*list of signals*>

where ***<list of signals>*** is a list of signals whose values are requested. For example:

```
>>: look list=a,sum
At Time= 23, Test=101:
a= '0' [Primary Input]
sum= '1' [EXOR]
```

Note that SIMIC generates a time/test stamp, and for each requested signal, outputs:

- The signal's name.
- The signal's value in SIMIC character format. If the driving device requires depths (switches, tristating elements, wire-ties, etc.), then the value's interval representation will also be displayed (in parenthesis). For example:

```
>>: look list=out
```

```
At time=48, Test=9:
out= 'Z' (32767/-32767) [TPADN]
```

- The signal's driver. This can be either a built-in or user-defined primitive, a primary input (or bus), or a global constant (logical-0, logical-1, Tristate (Z)).

The **LOOK** command's **LIST** keyword option is *not* sticky; SIMIC only reports the values of signals currently specified with the **LIST** keyword.

2.6.8.2 Displaying Topology As Well As Values

Very often, it is useful to be able to trace the source of an incorrect signal value. This process can be simplified by including some topological information in the **LOOK** output. The **INPUTS** (**IN**) keyword option controls reporting of element input values. To add fanin (element input value) information to the **LOOK** output, issue the command:

```
LOOK INPUTS:
```

To remove fanin information from the **LOOK** output:

```
NO LOOK INPUTS:
```

Once enabled, each subsequent **LOOK LIST** command will then contain information about the fanin values. For each input of the element generating the traced signal, this information consists of (1) the name of the signal connected to the input and (2) the input signal's value.

For example:

```
>>: look inputs:
>>: look list=carry-out
At Time= 0, Test= 1:
carry-out= '1' [OR]
I:= and1= '0' [AND]
I:= and2= '1' [AND]
I:= and3= '0' [AND]
```

In this example, the **carry-out** signal is a logical-1, and it is generated by an OR gate. This gate has three inputs, **and1**, **and2**, **and3**, each the output of an AND gate. The second input, **and2**, is causing the logical-1 at **carry-out**.

Note: if the signal being probed with the **LOOK** command is a wire-tie, then the state of each of the signal's drivers is *always* reported in fanin format, with each driver state reported as an "input" to the resultant wire-tie value:

```
>>: look list=sig_5
At Time= 42, Test= 9:
sig_5= 'H' (4/4) [WIRETIE]
I:= p1.1= 'H' (4/4) [UTGRP]
```

```
I:= p2.1= 'H' (6/6) [UTGRP]
I:= p3.1= 'Z' (32767/-32767) [UTGRN]
```

This LOOK output states that signal **sig_5** is a wire-tie of three drivers. Two of the drivers, the outputs of UTGRPs (unidirectional p-transistors, see Appendix A) with instance names **p1** and **p2** are driving logical-1 (at depths 4 and 6, respectively), while the third driver, the output of a UTGRN (unidirectional n-transistor, see Appendix A) with instance name **p3**, is tristating.

Fanout information can also be included in the **LOOK LIST** output with the **OUTPUTS** (**OU**) keyword option:

```
LOOK OUTPUTS:
```

Once enabled, each subsequent **LOOK LIST** command will then contain the instance names of all parts driven by the traced signal.

For example:

```
>>: look outputs: list=and2
At Time= 0, Test= 1:
and2= '1' [AND]
O:= or1 [OR]
```

which specifies that **and2** fans out to the OR gate named **or1**.

Fanout information can be subsequently removed from **LOOK** output with the command:

```
NO LOOK OUTPUTS:
```

2.6.8.3 Displaying All Signal States

The command:

```
LOOK LIST:
```

causes SIMIC to generate a table of all signals states (in alphanumeric sequence). The format of this table is equivalent to the **PRINT** (or **WRITE**) **dump format**, discussed in the Subsection *Selecting Signals to Output* in Chapter 2.4.

2.6.8.4 Displaying All Signals At A Specified State

Many times it is useful to see if there are any signals at an 'X' or 'Z' state. To display all signals at an 'X' state, use the command:

```
LOOK X:
```

Similarly, to display all signals at a 'Z' (tristate) value, use the **HIZ** (**HI**) keyword option:

```
LOOK HIZ:
```

The format of both reports is identical to the **LOOK LIST=** command's report format. These two commands do not honor the **INPUTS:** or **OUTPUTS:** keyword options, if in effect.

2.6.9 Forcing Signal States

2.6.9.1 Overview

SIMIC allows interactive modification of any signal's state by one of two methods, **SET** (**SE**) and **CLAMP** (**CL**). They differ only in the length of time that a signal value is forced. The **SET** value is maintained for a single test, after which the **SET** signal is freed to assume a consistent value (if the signal is an element output, its state must be consistent with the element's input state). A **CLAMP**ed value, however, will remain at the forced value from the specified test onwards, until explicitly freed with the **NO CLAMP** command.

2.6.9.2 Specifying Force Values And Tests

The value to force is selected by the keyword options: **ONE** (**ON**), **ZERO** (**ZE**), **X**, and **HIZ** (**HI**), for Logical-1, Logical-0, Unknown, and Tristate, respectively. The format is:

SET *<value-option>=<list of signals>*

or

CLAMP *<value-option>=<list of signals>*

where ***<value-option>*** is the appropriate keyword option, as described above.

For example:

```
SET ONE=a,carry-out
CLAMP ZERO=xyz
```

sets signals **a** and **carry-out** to logical-1, and **clamps** signal **xyz** to logical-0. These values will be applied at the start of the next test, after resumption of simulation. Signals **a** and **carry-out** will only remain forced for that test, but signal **xyz** will remain forced at logical-0 until explicitly released with the **NO CLAMP** command, or **CLAMP**ed to another value.

If values must be forced farther in the future than the next test, the **TNUM** (**TN**) keyword option of the **SET** or **CLAMP** run command can be used to specify the appropriate test. If the **TNUM** keyword is not specified, then the **SET** or **CLAMP** will be applied at the start of the next test. The form of the **TNUM** keyword option is:

SET TNUM=*<n>*

or

CLAMP TNUM=*<n>*

where ***<n>*** is the test number in which a value is to be **SET** or **CLAMP**ed.

The **TNUM** keyword option is *not* sticky, and if specified, must be included within the same command as the value assignments. For example, the sequence of commands:

```
CLAMP TNUM=100 ONE=abc,def
CLAMP ZERO=xyz
```

cause signals **abc** and **def** to be clamped to logical-1 at test 100, and signal **xyz** to be clamped to logical-0 at the next test.

If conflicting values are simultaneously specified for the same signal in **SET** and **CLAMP** commands, then the **CLAMP** command overrides the **SET**.

If a built-in flip-flop primitive's output is **SET** or **CLAMPED**, the master rank is also assigned the **SET** or **CLAMPED** value.

2.6.9.3 Cancelling Or Freeing Forced Values

The **NO** command prefix can be used to remove **SET** (or **CLAMP**ed) specifications at a particular value. Use the format:

NO SET *<value-option>=<list of signals>*

or

NO CLAMP *<value-option>=<list of signals>*

to selectively cancel previous **SET** or **CLAMP** specifications, or free previously **CLAMP**ed signals. Use the form:

NO SET *<value-option>*:

or

NO CLAMP *<value-option>*:

to cancel or free all forcing of signals to the specified value, ***<value-option>***, which is one of the four options: **ONE**, **ZERO**, **X**, or **HIZ**.

The **LIST** keyword may also be used in the **NO SET** or **NO CLAMP** command, to remove the **SET** or **CLAMP**, regardless of the injected value.

As with forcing signals, the operation of freeing signals is test-specific; a **SET** or **CLAMP** command, once issued, can only be explicitly removed from the test it is applied to. The **TNUM** keyword option can be used with the **NO** prefix to specify test number.

SIMIC queues user commands to force and free signal values This means that replaying the simulation will also replay the **SET** or **CLAMP** commands issued previously for this session.

<u>Examples</u>:

The command:

```
NO CLAMP LIST:
```

will remove all **CLAMP**s, either to **ONE**, **ZERO**, **X** or **HIZ**, assigned for the next test.

The sequence of commands:

```
SET ZERO=abc,def
SET TNUM=50 ZERO=abc
NO SET ZERO=abc
```

causes signal **def** to be set to logical-0 at the next test, and signal **abc** to be set to logical-0 at test 50. (The **NO SET** command cancels the request to set signal **abc** to logical-0 at the next test.)

The sequence of commands:

```
CLAMP TNUM=100 ONE=abc
NO CLAMP TNUM=120 ONE=abc
```

causes signal **abc** to be clamped to logical-1 at test 100 and subsequently released at test 120. The **LIST** keyword could also have been used in the **NO CLAMP** command.

2.6.10 Querying Delay Values

The current rise and fall delays of any signal in the circuit can be “queried”. This is accomplished with the **?DELAY** (**?DEL**) **LIST** (**LI**) command:

?DELAY LIST=<*list of signals*>

to “query” the delays of selected signals (in ***<list of signals>***), or

?DELAY LIST:

to “query” the delays for all signals.

2.6.11 Modifying Delay Values

2.6.11.1 Overview

In addition to supporting interactive selection of the timing set to be used for simulation, SIMIC also allows run-time modification of individual delays, either to an absolute value (e.g. 123) or a relative value (e.g +10%, -50%), by using the **FALL**, **RISE** or **CHANGE** keyword options of the **SET LIST** command. The **FALL** keyword modifies the FALL delays, the **RISE** keyword modifies the RISE delays, and the **CHANGE** keyword modifies both the RISE and FALL delays, of the signals specified by the **LIST** keyword.

2.6.11.2 Loading A Timing Set

SNL supports specification of three sets of driver delays, timing-check limits, and loading, corresponding to **TYPICAL** (**T**), **MINUMUM** (**MI**), and **MAXIMUM** (**MA**) values. The default set loaded by the **GET** (**GE**) command is **TYPICAL** values.

As described in Chapter 2.2, the timing set to be used for simulation can be selected with the **TIMING** (**TI**) keyword option of the **GET** run command. This option can actually be specified at any point in the simulation session. For example, to load the MINIMUM timing set, use the command:

```
GET TIMING=MINIMUM
```

2.6.11.3 Selecting Drivers For Delay Modification

The **LIST** (**LI**) keyword option for the **SET** (**SE**) command selects the drivers to be modified. The format is:

```
SET LIST=<list of signals>
```

to select the signals to be modified, ***<list of signals>*** is the list of signals, or

```
SET LIST:
```

to modify all signals.

If a signal has been specified that has multiple drivers (wire-tie), then all drivers will be modified accordingly.

2.6.11.4 Setting Delays To An Absolute Value

This feature was used in the Divide-by-7 example at the beginning of this chapter. It allows a driver's delay to be changed to a specified value (e.g. 3). The format is:

```
SET <delay>=<number>
```

where ***<delay>*** is either the **CHANGE** (**CH**), **RISE** (**RI**), or **FALL** (**FA**) keyword, and ***<number>*** is the new delay value. The **RISE** and **FALL** options may be specified in the same **SET** command, when they apply to the same signals.

For example:

```
SET CHANGE=1 LIST:
```

sets all rise and fall delays to unity.

As another example, the command:

```
SET FALL=25 LIST=abc
```

sets the fall delay of signal **abc** to 25 time-units, but does not modify its rise delay.

2.6.11.5 Setting Delays Relative To Their Current Value

This feature allows modification of delays by a percentage. The format is:

```
SET <delay>=+<percentage>
```

or

```
SET <delay>=<percentage>
```

to increase the specified driver's delays by a percentage, or

```
SET <delay>=−<percentage>
```

to decrease the specified driver's delays by a percentage, where ***<delay>*** is the appropriate **CHANGE**, **RISE**, or **FALL** keyword, and ***<percentage>*** is an integer, optionally followed by a percent sign (e.g., 50 or 50%), indicating the percentage change.

For example, to increase all rise delays by 30% and decrease all fall delays by 20%, issue the following command:

```
SET LIST: RISE=+30% FALL=-20%
```

If a negative percentage of -100% or less is specified, the designated delays are set to 0.

2.6.12 Querying Decay Values

SIMIC can be requested to report the current decay time of any signal in the circuit with the **?DECAY** (**?DEC**) **LIST** (**LI**) run command:

?DECAY LIST=<*list of signals*>

to “query” the decay value of selected signals (in ***<list of signals>***), or

?DECAY LIST:

to “query” the decay value for all signals.

2.6.13 Modifying Decay Values

2.6.13.1 Description

Decays are modified in the same manner as delays, described above, except the **DECAY** (**DE**) keyword is used instead of the **CHANGE**, **RISE**, and **FALL** keywords. In addition to absolute and relative modification, selected signals may be made to hold charge indefinitely when they tristate by specifying the value **INFINITE** (**I**) for the **DECAY** keyword. For example, to set all decays to infinite, use the command:

```
SET LIST: DECAY=INFINITE
```

2.6.14 Querying Signal Loading

SIMIC can be requested to report the current loading at any signal in the circuit with the **?LOADING** (**?LO**) **LIST** (**LI**) run command:

?LOADING LIST=<*list of signals*>

to “query” the loading at selected signals (in ***<list of signals>***), or

?LOADING LIST:

to “query” the loading at all signals.

2.6.15 Enabling And Disabling X-Propagation

SIMIC, by default, propagates an X whenever a spike, or part timing violation (setup, hold, or pulse-width) occurs. X-propagation is controlled by the **XPROPAGATE** (**XP**) command.

2.6.15.1 Spike Hazards

The **SPIKE** (**SP**) keyword option of the **XPROPAGATE** command controls X-propagation for spike hazards. To disable X-propagation whenever a spike occurs, issue the command:

```
NO XPROPAGATE SPIKE:
```

and to enable it:

```
XPROPAGATE SPIKE:
```

If spike propagation is disabled, then the tester interface file (with default file extension **tgn**) will not be generated. This is to insure that the design has been successfully run with spike propagation before committing the design to fabrication.

The criteria for generating an X-pulse and the size of the generated X-pulse can both be controlled on a per-signal basis. Run commands to modify X-pulse generation parameters are described below.

2.6.15.2 Near Hazards

The **NEAR** (**NE**) keyword option of the **XPROPAGATE** command controls X-propagation for near hazards. To enable X-propagation when a near hazard has been found, issue the command:

```
XPROPAGATE NEAR:
```

To disable X-propagation for near hazards:

```
NO XPROPAGATE NEAR:
```

By default, X-propagation is disabled for near hazards.

2.6.15.3 Functional Timing Violations

By default, X-propagation for timing check violations is enabled. That is, when a timing check violation occurs, the affected element signals are immediately set to X, and the X value is propagated to all element loads.

X-propagation for timing check violations can be controlled by specifying the **PART** (**PA**) keyword option and the appropriate timing check name(s) in the **XPROPAGATE** command. To disable X-propagation, issue the command:

```
NO XPROPAGATE PART=<part list> <timing-check>:
```

for specific parts, or:

`NO XPROPAGATE PART:` *<timing-check>*`:`

for all parts, where ***<part list>*** is a list of the parts to be affected, and ***<timing-check>*** is the timing check that should no longer cause X-propagation when a violation is detected. Different timing checks may be specified in the same command.

For example, assuming part **`f1`** is a DCF primitive, the command:

`NO XPROPAGATE PART=f1 HOLD: SETUP.NR:`

would disable X-propagation for all hold-time violations and for setup-time violations between the **`NR`** and **`CLK`** pins at this flip-flop. For information on the functional timing checks supported by each primitive, see Appendix A.

Subsequently, X-propagation can be re-enabled with the same commands, except the **`NO`** prefix would be omitted.

2.6.16 Querying Spike Control Parameters

2.6.16.1 Description

The current values of the spike generation parameters (**`FILTER`** and **`LIBERAL`** attributes) can be obtained for any signal in the circuit. This is done with the **`?SPIKE`** (**`?SP`**) **`LIST`** (**`LI`**) run command:

`?SPIKE LIST=`<*list of signals*>

to "query" the spike control parameter values of selected signals (in ***<list of signals>***), or

`?SPIKE LIST:`

to "query" the spike control parameter values for all signals.

2.6.17 Modifying Spike Control Parameters

2.6.17.1 Description

The **`LIBERAL`** (**`LIB`**) and **`FILTER`** (**`FILT`**) spike control parameters can be modified independently with the **`XPROPAGATE`** (**`XP`**) run command. The command:

`XPROPAGATE` <*keyword*>`=`<*value*> `LIST=`<*list of signals*>

changes the parameter ***<keyword>*** (**`LIBERAL`** or **`FILTER`**) to ***<value>*** for the selected signals in ***<list of signals>***, or

`XPROPAGATE` <*keyword*>`=`<*value*> `LIST:`

changes the parameter for all signals.

The ***<value>*** is a percentage from 0 to 100, and is specified as an integer, optionally followed by a percentage sign (e.g. 50 or 50%).

For example,

```
XPROPAGATE FILTER=20% LIST=abc,def
```

sets the **filter** spike control parameter of signals **`abc`** and **`def`** to 20%. See the Subsection *Controlling Spike Propagation* in Chapter 2.7 for descriptions of the **filter** and **liberal** spike control parameters.

To restore the spike control parameters to their original values (in the SNL description), use an asterisk (**`*`**) for the **filter** or **liberal** option. For example:

```
XPROPAGATE LIBERAL=* FILTER=* LIST:
```

would reset all **`FILTER`** and **`LIBERAL`** parameters to their original values.

2.6.18 Querying Functional Timing Check Settings

2.6.18.1 Description

SIMIC can be requested to report the current values of timing check parameters, and whether a **`WARN`**, **`BREAK`**, or **`XPROPAGATE`** command option is currently active for any primitive that supports these checks. The **`?CHECK`** (**`?CH`**) **`PART`** (**`PA`**) command:

```
?CHECK PART:
```

requests this information for all parts supporting timing checks, while the command:

?CHECK PART=<*list of parts*>

requests this information for the parts specified in ***<list of parts>***.

2.6.19 Modifying Functional Timing Check Parameters

2.6.19.1 Description

The **`SET`** (**`SE`**) command can be used to modify timing-check parameter values. The run command form:

SET PART: <*timing-check*>=<*value*>

modifies the specified timing check parameter values in all parts, and

SET PART=<*list of parts*> <*timing-check*>=<*value*>

does so for the parts selected in ***<list of parts>***.

The keyword, ***<timing-check>***, is the designated timing-check name (e.g., **`SETUP`**, **`HOLD`**, etc.), and ***<value>*** is the specified value, either absolute or relative, as described below.

For information on the timing-checks supported by SIMIC primitives, see Appendix A.

2.6.19.2 Setting Timing Check Parameters To Absolute Values

This feature allows a timing check parameter to be changed to a specified value (e.g. 3). In this case ***<value>*** is a positive integer. For example:

```
SET PART=F1 PW.C.L=10
```

assigns the value 10 time-units to the pulsewidth check parameter for low clock at the memory element named **`F1`**.

2.6.19.3 Setting Timing Check Parameters Relative To Current Values

This feature allows modification of timing check parameters by a percentage. The format for value is:

+<*percentage*>

or

<*percentage*>

to increase the specified timing check parameter by a percentage, or

−<*percentage*>

to decrease the specified timing check by a percentage.

The ***<percentage>*** specification is an integer, optionally followed by a percent sign (e.g. 50 or 50%), indicating the percentage change.

For example, to increase all setup timing checks by 30% and decrease all hold timing checks by 20%, issue the following command:

```
SET PART: SETUP=+30% HOLD=-20%
```

2.6.20 Replaying Portions of the Simulation

2.6.20.1 Description

The **`RESTORE`** (**`RE`**) command loads an initial circuit state for subsequent simulation. The loaded state is specified by the **`RESTORE`** command's **`TNUM`** (**`TN`**) keyword option. SIMIC allows the simulation state to be restored to:

1. The uninitialized state at the start of simulation. This is accomplished with the command:

   ```
   RESTORE TNUM=0
   ```

2. A state previously saved in a checkpoint file. This is accomplished with the command:

   RESTORE TNUM=<*n*>

   where ***<n>*** is a test that has been checkpointed (circuit state was saved). The checkpoint file can be explicitly specified with the **`FILE`** (**`FI`**) keyword:

   RESTORE FILE=<*filename*>

If unspecified, the checkpoint file is assumed to have the default file name and the default extension **sav**.

3. The last stable state. This is accomplished with the command:

```
RESTORE TNUM=*
```

This option allows a condition to be replayed as many times as necessary to determine why the condition occurs. This feature was used in the Divide-by-7 debugging example at the beginning of this chapter.

2.6.20.2 Creating The Checkpoint File

The **SAVE** (**SA**) command directs SIMIC to save checkpoint states to a saved-state file. If unspecified, this file has the default file name and the default extension **sav**. The saved-state file can be explicitly specified with the **FILE** (**FI**) keyword:

SAVE FILE=<*filename*>

Only completely stable (no pending decays or other circuit activity) points may be saved into this file. Once saved, this state can be retrieved, and simulation can be continued from this point. The **PSTEP** (**PS**) keyword option of the **SAVE** command initiates saving checkpoints and controls the save interval. The command form is:

SAVE PSTEP=<*n*>

where ***<n>*** is the number of stable points between saves. To disable checkpointing, use the command:

```
NO SAVE PSTEP:
```

SAVE operations can be restricted to a specified test (patterns) or time (waveforms) interval with the **PRANGE** option of the **SAVE** command. See the Section *Restricting Simulation Options To A Specified Simulation Interval* covering **PRANGE** specifications earlier in this chapter.

A special option of the **SAVE PSTEP** command is useful for interactive debugging:

```
SAVE PSTEP=0
```

This will save the *current* state into the checkpoint file, but will not modify the checkpointing interval (***<n>*** above) if previously specified. To insure that the circuit is stable, issue the command:

```
RESTORE TNUM=*
```

prior to **SAVE** command.

To display which states have been saved in a file, use the command:

```
RESTORE TNUM=?
```

and optionally the **FILE** keyword option (to specify the file name) if the file does not have the default file name.

2.6.20.3 Restoring The Saved State's Time And Test

There are two uses for the checkpoint file:

1. Incremental simulation.

   Here, a circuit state is restored and a new stimulus sequence is to be applied. In this case, the state of the network should be restored, but not the time and test at which the **save** occurred. This is useful when experiments are being conducted with different stimulus sequences that require the restored state as their initial state.

   To restore the state, but not the time and test, use the command sequence:

   ```
   NO RESTORE PRANGE:
   RESTORE TNUM=<n>
   ```

2. Resimulation.

   Here, a circuit state is to be restored and the same stimulus sequence is to be applied. In this case, the state of the network, the time, and the test at which the **save** occurred must all be restored. This is the default operation in SIMIC. If you have disabled this mode with the:

   ```
   NO RESTORE PRANGE:
   ```
   command, you can re-enable it with the:

   ```
   RESTORE PRANGE:
   ```
   command. Then, issue the command:

   ```
   RESTORE TNUM=<n>
   ```

Chapter 2.7   Circuit Modeling

2.7.1   Introduction

SIMIC supports many features that facilitate circuit description, allow accurate delay modeling and rapid detection of timing problems. This chapter describes how to represent circuits hierarchically, describe delay and loading characteristics, and create cell libraries. It also describes how SIMIC handles special circuit configurations, such as wire-ties and paralleled elements.

Design verification is often performed as a two-step process. The first step is to verify functionality, that is, making sure that the logical design is correct and complete. Addressing timing problems at this point would unnecessarily complicate the task. With this goal successfully accomplished, the second step is to detect and correct timing problems. Following this methodology requires that the user be able to control the degree of timing checks and pessimism introduced during simulation. This chapter describes how to accomplish this control from the SNL circuit description. Much of this control is also available with SIMIC run commands issued during simulation (see Chapter 2.6).

Whenever a new SNL keyword is introduced in this chapter, its valid abbreviations (if any) are also given in parentheses.

2.7.2   Hierarchical Description

SNL supports hierarchy in a regular manner. As described in Chapter 1.2, each PART statement instantiates a component and connects it to other components in a type block. The components can be built-in primitives, user-defined primitives, or macros (structural descriptions of subcircuits containing primitives and/or other macros; the lowest level macros contain only primitives). The type of component is specified by the PART statement's **`TYPE`** keyword field.

In general, a SNL description contains one or more **type blocks**, each defining a macro or BOOLEAN subcircuit. All type blocks begin with a TYPE statement that defines the subcircuit's pins and their electrical characteristics. In a macro, PART statements follow the TYPE statement to describe its internal structure. An entire circuit description can span multiple files.

A macro's depth, or level of nesting, is unrestricted.

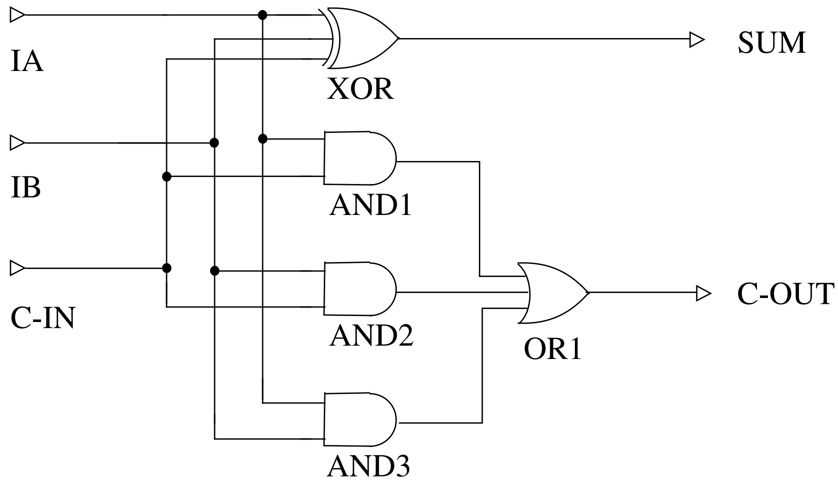


Figure 2.7-1(a) Full-Adder Circuit

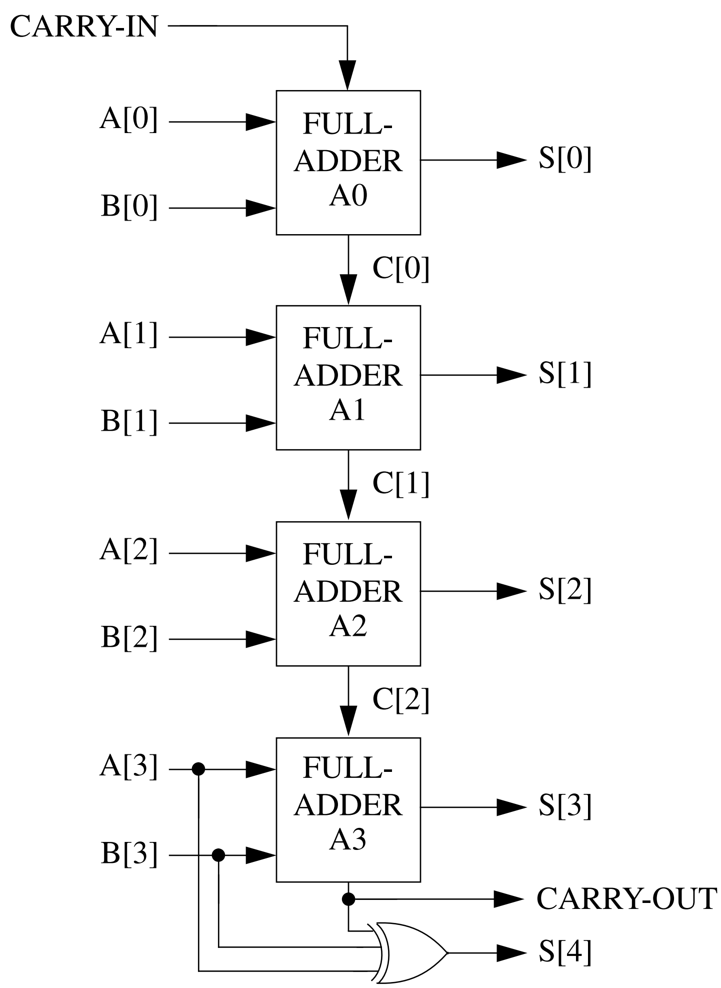


Figure 2.7-1(b) Four-Bit Adder

2.7.2.1 Instantiating Macros

A four-bit ripple-carry adder will be used as an example to illustrate hierarchical SNL descriptions. Figure 2.7-1(a) illustrates the one-bit full-adder circuit of Chapter 1.1, which is used here as the basic adder cell, and Figure 2.7.1(b) illustrates the four-bit adder's block diagram. (The exclusive-or gate and its output signal, **s[4]**, are not really part of the four-bit adder structure; this element performs sign extension of the four-bit inputs to produce the proper sign bit of a five-bit sum.)

The SNL descriptions of both the full-adder and the four-bit adder are shown in Figure 2.7-2.

```
!format p=  t=  i=  o=

type=full-adder  i=ia,ib,c-in  o=sum,c-out
   xor     exor    ia,ib,c-in       sum
   and1    and     ia,c-in          and1
   and2    and     ib,c-in          and2
   and3    and     ia,ib            and3
   or1     or      and1,and2,and3   c-out


type=four-bit   i=a[3:0],b[3:0],carry-in  $
                o=s[4:0],carry-out
%declare integer2 = a[3:0],b[3:0],s[4:0]
   a0   full-adder   a[0],b[0],carry-in  s[0],c[0]
   a1   full-adder   a[1],b[1],c[0]      s[1],c[1]
   a2   full-adder   a[2],b[2],c[1]      s[2],c[2]
   a3   full-adder   a[3],b[3],c[2]  s[3],carry-out
   s4   exor         a[3],b[3],carry-out      s[4]
```

Figure 2.7-2 SNL Description of Four-Bit Adder

The first point to note is that the PART statements instantiating the full-adders are structurally no different from the PART statements that instantiate primitives. The second point is that the number of inputs and outputs of each instantiated full-adder agrees with the number of inputs and outputs of the **full-adder** macro, since there is a one-to-one correspondence. For example, part **a1** in the **four-bit** macro has three inputs, **a[1]**,**b[1]**,**c[0]** that are connected to pins **a**,**b**,**carry-in**, respectively, of the instantiated **full-adder** macro, and two outputs, **s[1],c[1]**, that are connected to the macro's output pins, **sum**,**carry-out**.

Instances of internal macro parts and signals must be distinguished from the corresponding parts and signals of other instances of the same macro. In

Hierarchical names of internal macro parts and signals are formed by prefixing their original names with the instantiating PART statement's pathname.

SIMIC, they are distinguished by prefixing each internal part's and signal's name with the **pathname** of the instantiating PART statement. The pathname is constructed by concatenating all part names in the order encountered from the highest level to the current part, delimiting adjacent names with dots (.). For example, the full-adder instance named **a1** has five internal parts named **a1.xor**, **a1.and1**, **a1.and2**, **a1.and3**, and **a1.or1**. Similarly, its three internal signals are named **a1.and1**, **a1.and2**, and **a1.and3**. The remaining signals of the **full-adder** are connected to the macro's pins, and serve as "parameters" or "dummy variables"; their names are replaced by the names of the signals connected to the corresponding macro instance pins. In summary, the macro instance:

```
a1   full-adder   a[1],b[1],c[0]   s[1],c[1]
```

is expanded (flattened) into

```
a1.xor    exor    a[1],b[1],c[0]   s[1]
a1.and1   and     a[1],c[0]        a1.and1
a1.and2   and     b[1],c[0]        a1.and2
a1.and3   and     a[1],b[1]        a1.and3
a1.or1    or   a1.and1,a1.and2,a1.and3   c[1]
```

All types in the expanded macro are primitives. If, however, the part named **a1.and2** was a macro instead of an AND gate, all its internal part and signal names would be prefixed with the pathname **a1.and2**.

2.7.2.2 Main Type

Referring to Figure 2.7-2, there is no structural difference between the **full-adder** and the **four-bit** type blocks—each consists of a type statement followed by part statements. Either type block can be simulated directly (of course, compatible stimuli would have to be defined for the macro selected for simulation). The type specified in the **GET TYPE** command is the circuit that is actually loaded and compiled. This type is called the **main type**. Thus, the command:

```
get type=full-adder
```

would cause the full-adder to be simulated, while the command

```
get type=four-bit
```

would cause the four-bit adder to be simulated.

2.7.2.3 Sample Simulation of the Hierarchical Circuit

Except for the fact that some part and signal names are hierarchical, simulation of a hierarchically-described circuit is exactly the same as that for a flat description.

Figure 2.7-3 illustrates a run file for the four-bit adder. After compiling this macro with the **GET TYPE** command, pattern stimuli are defined for its inputs. **a**, **b**, and **carry-in**. The patterns for the four-bit arrays **a** and **b**

```
define file=adder
get type=four-bit
define pa.4.int = 7  7  7  7 -7 -7 -7 -7
define pb.4.int = 5  5 -5 -5  5  5 -5 -5
define pc.1     = 0  1  0  1  0  1  0  1
apply patterns=pa list=a
apply patterns=pb list=b
apply patterns=pc list=carry-in
print list=a*b*carry-in**carry-out*s***$
     a[3],a[2],a[1],a[0]*b[3],b[2],b[1],b[0]$
    *s[4]s[3],s[2],s[1],s[0]
simulate
look list=a1.and2
quit
```

Figure 2.7-3(a) adder.run File For the Four-Bit Adder

are defined in INT (integer) format, since the circuit performs an arithmetic function. This is the also reason that the circuit's inputs, **a** and **b**, and its output, **s**, were defined as arrays in the SNL description (Figure 2.7-2), rather than as individual signals. This, combined with the **%DECLARE** statement in the four-bit macro (which specifies that **a**, **b**, and **s** be treated as 2's complement numbers in any **PRINT** or **WRITE** command), means that the **PRINT** statement of Figure 2.7-3 will cause **a**, **b**, and **s** to be printed out as 2's complement numbers. This **PRINT** statement also specifies that the individual components of **a**, **b**, and **s** be printed, as binary signals, for comparison with the integer values.

The patterns in this example are simple; all possible combinations of operand signs and carry-in values when the absolute magnitudes of **a** and **b** are 7 and 5, respectively. After simulation, the **LOOK** command is used to examine an arbitrarily-picked hierarchically-named signal, **a1.and2**, to show they are referenced in the same manner as non-hierarchical signals.

Figure 2.7-3(b) shows the corresponding simulation session. SIMIC was directed to the above run file with the **EXECUTE** command. Note that the arithmetic values reported for **s** are correct for the given values of **a**, **b**, and **carry-in**. Also note that these numbers are the correct interpretation of 2's complement representation for the four-bit arrays **a** and **b**, and the five-bit array **s**.

```
The SIMIC Logic simulator... Version 1.00.00
Genashor Corp, Copyright 1991
Main Get Network : FOUR-BIT
GET completed, Circuit totals:  Parts = 21; Signals = 32
   Inputs = 9; Busses = 0; Outputs = 6
>>: execute file=adder
Remark= Options: (Fault Free simulation)
Remark=   Pattern stimuli, Near Filter, Spike Filter
Remark=   Stable Before Decay, Dynamic Delay

C=                 A  B C  C   S   AAAA BBBB SSSSS
C=                      A  A       [[[[ [[[[ [[[[[
C=                      R  R       3210 3210 43210
C=                      R  R       ]]]] ]]]] ]]]]]
C=                      Y  Y
C=                      -  -
C=                      I  O
C=                      N  U
C=                         T

        0 T     1:  7  5 0  0  12   0111 0101 01100
        0 T     2:  7  5 1  0  13   0111 0101 01101
        0 T     3:  7 -5 0  1   2   0111 1011 00010
        0 T     4:  7 -5 1  1   3   0111 1011 00011
        0 T     5: -7  5 0  0  -2   1001 0101 11110
        0 T     6: -7  5 1  0  -1   1001 0101 11111
        0 T     7: -7 -5 0  1 -12   1001 1011 10100
        0 T     8: -7 -5 1  1 -11   1001 1011 10101
At Time= 0, Test= 8:
A1.AND2= '1' [AND]

Quit Command Issued... Leaving SIMIC

Total SIMIC CPU-time = 0.46 sec. (00:00:00.46)
............... User : 0.19 sec. (00:00:00.19)
............. System : 0.27 sec. (00:00:00.27)
........ Page faults : 56
```

Figure 2.7-3(b) SIMIC Simulation of the Four-Bit Adder Using An EXECUTED Run File

2.7.3 Modeling Delays

In SNL, delays may be specified either locally, in the PART or TYPE statement, or globally, in a **delay table**. Delays may depend on signal loading. Delay tables are specified in the **!DELAY** section of the SNL description, while pin and net loading are specified in PART and TYPE statements of the **!LOGICAL** section. This section describes these SNL constructs.

The SIMIC compiler adds all pin loading on each net (i.e., the loading at each element pin connected to the net) to the net's contribution (wiring capacitance) to obtain the total net loading. It then references the driver delay characteristics to compute the rise and fall delays of the net's driver(s).

Delays can also be changed at run time using the **SET** command.

2.7.3.1 SIMIC Time-Units

In SIMIC, a time-unit is the smallest quantum of time that can be processed. All SIMIC delays are expressed in time-units. Time-units are normalized time measurements; they can represent any real-time value (e.g., picoseconds, nanoseconds), depending on the selected scale factor used to define delays. Care must be taken in this selection; if a technology has gate delays in the nanosecond range, and delays have been scaled so that 1 time-unit represents 1 picosecond, then each signal delay will be thousands of time-units. During simulation, cumulative time will increase rapidly. Since SIMIC maintains elapsed simulated time internally as a 32 bit integer, the maximum possible simulation time is 2,147,483,647 time-units. Large internal delays would unnecessarily limit the number of stimuli that can be simulated before maximum time is reached.

The correspondence between time-units and real-time can be optionally entered in the **!DELAY** statement's **TIME-UNITS** keyword-field. For example:

```
!DELAY time-units=1e-9
```

specifies that one time-unit corresponds to one nanosecond. If this correspondence is specified, SIMIC reports its value at run time.

2.7.3.2 Delay Curves

Delays are specified with fixed-point or floating-point numbers having up to six significant digits. A fixed-point number is a decimal number, possibly containing a decimal point (a rightmost decimal point is implicit for integers). A floating-point number is a fixed-point number multiplied by an integral power of 10; the letter "E" (or "e") separates this exponent from the fixed-point number. The plus (+) sign for positive exponents is optional. For example, the number 72 can be represented as:

```
72, 72., 7200e-2, .0072e4, etc.
```

Delays may vary linearly with loading, as shown in Figure 2.7-4. The delay vs. loading relation in this figure can be described as either:

1. A line that goes through the coordinates (2,4) and (6,8), or,
2. A line that has a y-intercept of 2 and a slope of 1.

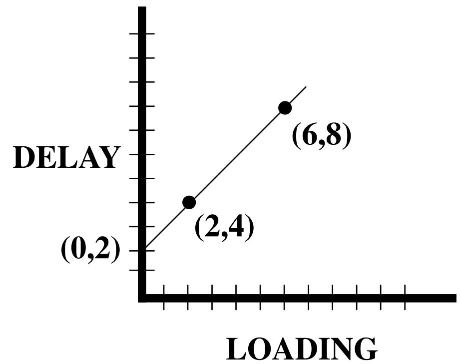


Figure 2.7-4 Typical Delay vs. Loading Relation

The two-point form is:

`(`*<load1>*`,`*<delay1>*`)` `(`*<load2>*`,`*<delay2>*`)`

In this example, the two-point representation would therefore be:

```
(2,4)(6,8)
```

which defines a delay whose value is 4 when the loading is 2, and 8 when the loading is 6. Optional whitespace may be placed between the right parenthesis of the first point and the left parenthesis of the second.

The intercept-slope form is:

`[`*<intercept>*`,`*<slope>*`]`

In this example, the intercept-slope representation would be:

```
[2,1]
```

Two-point and intercept-slope forms may be used interchangeably.

Constant delays are a special case, with zero slope. For example, a constant delay of 4 could be represented as:

`4` or `(1,4)(10,4)` or `[4,0]` etc.

2.7.3.3 Global Delays

Global delays are specified in the **!DELAY** section of the SNL description. Rise and fall delays are independently specifiable. Each global delay characteristic is assigned a unique name that is referenced by individual PART and TYPE statements in the **!LOGICAL** section. Each global delay definition has the form:

```
DELAY=<name> RISE=<delay> FALL=<delay>
```

for delay characteristics having different rise and fall delays, or

```
DELAY=<name> CHANGE=<delay>
```

for delay characteristics having identical rise and fall delays. In both forms, ***<delay>*** is a delay specification. For example:

```
DELAY=del5 RISE=[2,1] FALL=7
```

defines a global delay having a rise delay characteristic as shown in Figure 2.7-4 and a constant fall delay of 7, independent of loading.

The <***delay***> specification may contain minimum and maximum delays as well as typical delays. Its complete format is:

<typical-delay>;*<minimum-delay>*;*<maximum-delay>*

where each delay specification is either a two-point or intercept-slope description. For example:

```
DELAY=del12_8_15  CHANGE=[12,1];8;(0,15)(2,19)
```

The typical delay must be specified. If either the minimum or maximum delay is omitted, it defaults to the typical delay.

Delays are assigned to TYPE and PART outputs with the **OUTPUT-DELAY (ODEL)** keyword-field, and to busses with the **BUS-DELAY (BDEL)** keyword-field. The value part of these keyword-fields, which contains the global delay names, is in a one-to-one correspondence with the respective outputs and busses. For example,

```
P=a2  T=full-adder  I=a[2],b[2],c[1] $
    O=s[2],c[2]  ODEL=del5,del12_8_15
```

assigns the global delay **del5** to output **s[2]** and **del12_8_15** to **c[2]**.

If a signal's delay is unspecified, the delay defaults to 0. Thus, if the first delay in the above PART statement were omitted:

```
P=a2  T=full-adder  I=a[2],b[2],c[1] $
     O=s[2],c[2]  ODEL=,del12_8_15
```

the rise and fall delays of signal **s[2]** default to 0. Note the placeholder comma preceding the delay name **del12_8_15**. If this comma were missing, then delay **del12_8_15** would be assigned to **s[2]** and the rise and fall delays of **c[2]** would default to 0.

As mentioned above, the global delay definitions are specified in the **!DELAY** section, and are referenced by PART and TYPE statements in the **!LOGICAL** section:

```
!DELAY time-units=1e-9
     .........
DELAY=del5 RISE=[2,1] FALL=7
DELAY=del12_8_15  CHANGE=[12,1];8;(0,15)(2,19)
     .........
     .........
     .........
!LOGICAL
     .........
 P=a2  T=full-adder  I=a[2],b[2],c[1] $
     O=s[2],c[2]  ODEL=del5,del12_8_15
     .........
```

The **!LOGICAL** section could be in the same network description file as the **!DELAY** section, or it could be in a different file. Delays defined in a **!DELAY** section are truly global; that is, available to PART and TYPE statements in all network description files explicitly selected with the **GET FILE** keyword option or referenced by an **!INCLUDE** statement. Furthermore, the global delay tables can be contained in multiple **!DELAY** sections, possibly in different files. Regardless of how the delay tables are organized, however, global delay names must be unique. Assigning the same name (e.g., **del5**) to two *different* delay characteristics is a fatal error.

Global delays are very convenient when constructing simulation libraries for variants of a particular technology. Here, the functions of the logic cells and loading characteristics are identical, but each variant requires a different set of delay-vs.-loading curves. This is accomplished very simply by creating separate files, each containing a **!DELAY** section for one variant, and then referencing the appropriate delay table file with the **GET FILE** option.

2.7.3.4 Local Delays

Delays may also be assigned locally, within a PART or TYPE statement, without referencing a named global delay. Local delays are unnamed; they are specified directly. This is especially useful for generating SNL descriptions with automated netlisters. The format for defining local delay characteristics is identical to the ***<delay>*** format, described above, for the **DELAY** statement's **RISE**, **FALL**, and **CHANGE** keywords. The corresponding keywords for PART and TYPE busses are **BUS-RISE (BRISE)**, **BUS-FALL (BFALL)**, and **BUS-CHANGE (BCHANGE)**. For outputs, the corresponding keywords are **OUTPUT-RISE (ORISE)**, **OUTPUT-FALL (OFALL)**, and **OUTPUT-CHANGE (OCHANGE)**.

For example, assume that a type block named **my_type** is being defined for a subcircuit that has three input pins, two bidirectional pins, and two outputs pins. Then the TYPE statement:

```
T=my_type  I=a,b,c  B=d,e  O=f,g  BCHANGE=3,4  $
  ORISE=(5,3)(20,4)  OFALL=[3,1]  OCHANGE=,2
```

assigns:

bus **D** rise and fall delays of 3
bus **E** rise and fall delays of 4
output **F** a rise delay of (5,3)(20,4) and a fall delay of [3,1]
output **G** rise and fall delays of 2.

2.7.3.5 Specifying Pin Loading

Loading can be specified for all pins of a TYPE statement and all pins of an instantiated component in a PART statement. A load value can also be specified for each signal, typically representing wiring capacitance. Load values are specified as fixed-point or floating-point numbers. If no loading is specified for a signal or a pin, then the loading value is defaulted to 0.

Like delays, the general format for specifying loading is a three-tuple of values—typical, minimum, and maximum—separated by semicolons.

The keywords for specifying pin loads in PART and TYPE statements are **BUS-LOADS (BLOD)**, **INPUT-LOADS (ILOD)**, and **OUTPUT-LOADS (OLOD)** for bus, input, and output pins, respectively. The specified values are in a one-to-one correspondence with the respective signals. For example, adding loading to the above TYPE statement for **my_type**:

```
T=my_type  I=a,b,c  B=d,e  O=f,g  BCHANGE=3,4  $
  ORISE=(5,3)(20,4)  OFALL=[3,1]  OCHANGE=,2  $
   ILOD=1,,3 BLOD=5;4;6,7.89 OLOD=10,11
```

assigns:

input pin **A** a load of 1
input pin **B** a load of 0 (since no loading is specified)

input pin **C** a load of 3
bus pin **D** typical,minimum,maximum loading of 5;4;6
bus pin **E** a load of 7.89
output pin **F** a load of 10
output **G** a load of 11.

Nets are assigned loading by instantiating the built-in LOAD element:

`P=`*<name>* `T=load O=`*<net-name>* `OLOD=`*<value>*

or

`P=`*<net-name>* `T=load OLOD=`*<value>*

where:

- ***<net-name>*** is the hierarchical name of the net
- ***<value>*** is the load value assigned to the net.

In the first form, with the **OUTPUT** keyword-field explicitly specified, the assigned part name, ***<name>***, is *arbitrary* (this is the only case where a part name has no significance).

As an example of the second form, the statement:

```
PART=a.b.c  TYPE=load  OLOD=3;2;5
```

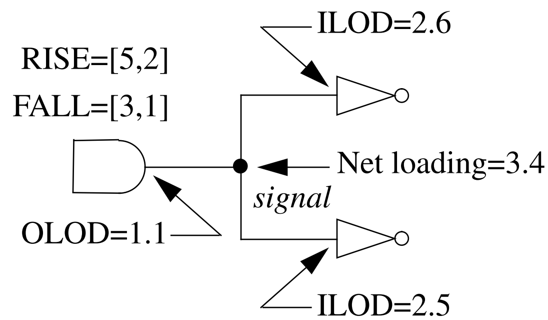
assigns typical,minimum,maximum loading of 3,2,5, respectively, to the signal named **a.b.c**.

2.7.3.6 Resultant Delays

When a load-dependent driver delay is specified, either by reference to a global delay or by a local delay, the SIMIC compiler totals all loading associated with the driven net to obtain the resultant delay from the specified delay vs. loading characteristic. Calculations are performed in floating point, and the results rounded to the nearest integer.

For example, in the circuit shown in Figure 2.7-5, the total loading on the net named **signal** is (1.1 + 2.6 + 2.5 + 3.4 =) 9.6. Thus, the AND gate's rise delay is 24 (5 + 2×9.6 = 24.2 rounded down), and its fall delay is 13 (3 + 1×9.6 = 12.6 rounded up).

If the result after interpolation is negative, SIMIC sets the delay to 0.




```
P=driver t=and i=a,b o=signal orise=[5,2] $
   ofall=[3,1]  olod=1.1
p=load1  t=inv  i=signal  ilod=2.6
p=load2  t=inv  i=signal  ilod=2.5
p=signal t=load olod=3.4
```

Figure 2.7-5 Delay Computation Example

2.7.3.7 Delays At Paralleled Elements

Drivers are sometimes paralleled to decrease rise and fall delays for heavily-loaded busses. Usually, the drivers are either identical or at least close in their drive capabilities. The SIMIC compiler considers elements paralleled when (1) their outputs are tied together, (2) they are all the same type, which must be one of the following built-in primitives: INV, AND, NAND, OR, NOR, BTGRN, BTGRP, UTGRN, or UTGRP, and (3) they have the same inputs (though not necessarily in the same order). When these criteria are met, SIMIC replaces the paralleled elements with a single element whose delay characteristic has:

1. a y-intercept that is the minimum y-intercept of all the paralleled elements, and
2. a slope obtained by treating the slope of each element's delay vs. loading characteristic as a resistance and computing the equivalent resistance of the parallel combination

The rise and fall delay constructions are performed independently. Figure 2.7-6 illustrates this replacement.

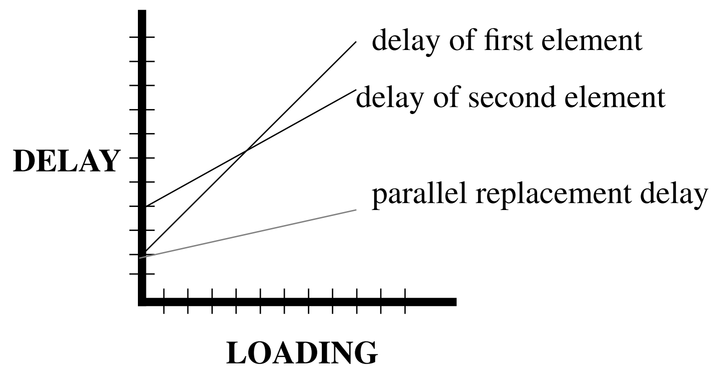


Figure 2.7-6 Illustration of Parallel Element Delay Reduction

2.7.3.8 Modifying Delays At Run Time

The **SET** command can be used to modify delays at run time. This is useful for circuit debugging and experimentation. The options that can be specified are:

1. **<n>** - set delay to the specified value, **<n>** (an integer)
2. **+<p>%** or **-<p>%** - increase or decrease delay by the specified percentage, **<p>** (an integer). (Note: the + sign is optional)

The **RISE**, **FALL**, and **CHANGE** keywords are used to specify that rise delays, fall delays, or both, respectively, be modified in the requested manner. The **LIST** keyword specifies which signals should be affected. For example,

```
SET RISE=30 LIST=a,b,c
```

sets the rise delay of the three signals to 30, while

```
SET CHANGE=-10% LIST:
```

reduces all delays by 10% (note that specification of a percentage less than, or equal to, -100% will set the affected delays to 0).

The Section *Modifying Delay Values* in Chapter 2.6 contains a more complete description of interactive delay modification.

2.7.4 Decays

By default, SIMIC instantaneously sets a floating net to the **Z** state, which represents an unknown value at floating strength. Decay times can be set to non-zero values on a per-node basis either in the circuit's SNL description of by using the **SET DECAY** run command (see the Section *Modifying Decay Values* in Chapter 2.6).

When a node has a non-zero decay time, its value instantaneously changes to **C**, representing logic 1 at floating strength, if it was previously a driven 1, or to **D**, representing logic 0 at floating strength, if it was previously a driven 0. The node will remain at these values until it decays to **Z** (or it is driven again).

2.7.4.1 Specifying Decays In SNL

Decay characteristics are specified in PART or TYPE statement with the **BUS–DECAY (BDEC)** keyword-field for busses or the **OUTPUT–DECAY (ODEC)** keyword-field for outputs. The specified values for these keywords have a one-to-one correspondence with the busses and outputs, respectively. The decay values that can be specified are:

1. a positive number specifying the decay time
2. the name of a global delay
3. the reserved word **INFINITE**, or any prefix of this word.

The first and third options are self-explanatory. The second option, specifying a global delay, allows the decay time to depend on loading (capacitance). Since delay characteristics specify two delays, rise and fall, the node's decay time is taken as the average value of the two delays for the given node capacitance. Generally, it is easiest to represent load-dependent decays by creating special global delays that contain the **CHANGE** keyword-field.

For example, the TPADN, a built-in primitive, is a tristating element with two inputs, EN (enable) and D (data). When EN is logical-1, the output is equal to the data input, and when EN is logical-0, the output tristates. Given the PART statement:

```
p=tri t=tpadn i=enable,data5 o=out5 odec=del83
```

suppose that, for the capacitance at the net named **out5**, the rise and fall delays for the delay characteristic **del83** are 90 and 110, respectively, Then the decay time for this net will be their average value, 100.

Note: if different decay times are specified for a signal that has multiple drivers (wire-tie), then the signal's decay will be the minimum value specified for any of its drivers.

2.7.4.2 Modifying Decays At Run Time

The **SET DECAY** command can be used to modify decays at run time. The **DECAY** keyword supports the same three options as the **SET** command's delay modification keywords, plus a fourth option, setting infinite decay:

1. **<n>** - set decay to the specified value, **<n>** (an integer)
2. **+<p>%** or **-<p>%** - increase or decrease decay by the specified percentage, **<p>** (an integer). (Note: the + sign is optional)
3. ***** - restore original decay from SNL description
4. **INFINITE** - set decay to infinite (any prefix of this word is valid)

The **LIST** keyword specifies the signals to be affected by the selected option. For example:

```
SET  DECAY=infinite  LIST=a,b
```

sets the decays of the two signals to infinite.

2.7.5 Input High Impedance Default

By default, the value Z, unknown value at floating strength, is treated as an X at (unidirectional) element inputs that are not strength-sensitive. (Of course, a signal's strength may be very important at bus pins, where the signal may be wire-tied. and flow may be bidirectional.)

This default is applicable to many modern technologies. For example if a CMOS inverter's input is floating, its output value will be uncertain. However, it is not correct for all technologies. In current-mode logic, a floating signal may be equivalent to 0. In the old DTL technology, a floating signal is equivalent to 1 at an AND gate input, and to 0 at an OR gate input.

If the default is inappropriate, the logical equivalent of Z may be explicitly specified in PART and TYPE statements, on a per-input basis, using the **INPUT-HIZ (IHIZ)** keyword-field. For example, the PART statement:

```
part=d  type=and  i=a,b,c  ihiz=1,0
```

specifies that a floating unknown value should be treated as logical-1 at input **a**, and as logical-0 at input **b**. Since no **IHIZ** value is specified for input **c**, a floating unknown at this input will be treated as X.

2.7.6 Verifying Timing Tolerances

When simulation is performed to verify logical correctness and completeness, timing considerations are deferred. At this time, the effects transient input states and narrow margins between events are ignored. However, in the final phases of design verification, these situations must be detected and corrected to eliminate potential timing problems.

2.7.6.1 Functional Timing Checks

Timing checks are supported for the DNL and DPL (DL) latch and the DNCF, DPCF (DCF), JKNCF (JKCF), JKPCF, TNCF (TCF), and TPCF edge-triggered flip-flops. These checks can be specified within a **`TIMING–CHECKS`** block in any PART statement instantiating these built-in primitives.

In the following description, the *active clock edge* is the clock transition that causes the latch or flip-flop output to change state. This edge is the rising clock transition for the DPL (DL), DPCF (DCF), JKPCF, and TPCF primitives, and the falling clock transition for the DNL, DNCF, JKNCF (JKCF), and TNCF (TCF) primitives.

The supported timing checks are:

1. **`SETUP`** – this check specifies the minimum duration that an input must be stable *prior to* an active clock edge:
   - DNL, DPL, DNCF, DPCF – setup from D (**`SETUP.D`**)
   - JKNCF, JKPCF – setup from J (**`SETUP.J`**) and setup from K (**`SETUP.K`**)
   - Additionally, all eight primitives support setup from reset (**`SETUP.NR`**), and setup from set (**`SETUP.NS`**). These setup times represent the minimum duration that the reset (set) must be *inactive* to reliably set (reset) the memory element via clock
2. **`HOLD`** – this check specifies the minimum duration that an input must be stable *after* an active clock edge:
   - DNL, DPL, DNCF, DPCF – hold to D (**`HOLD.D`**)
   - JKNCF, JKPCF – hold to J (**`HOLD.J`**) and hold to K (**`HOLD.K`**)
   - Additionally, all eight primitives support hold to reset (**`HOLD.NR`**), and hold to set (**`HOLD.NS`**)
3. **`PULSE–WIDTHS –`** this check specifies the minimum width of a pulse on the set, reset, or clock lines. All eight primitives support: pulse-width reset (**`PW.NR`**), pulse-width set (**`PW.NS`**), and high and low pulse-width clock (**`PW.C.H`** and **`PW.C.L`** respectively).

The setup and hold checks for the asynchronous, active-low, set and reset inputs of all eight primitives are associated with the time duration between the rising (trailing) edge of pulses on these inputs and the active clock edge.

Unspecified timing checks default to 0 (disabled).

The syntax of a **TIMING–CHECKS** block is:

```
TIMING–CHECKS=begin; <spec>; …;<spec>; end;
```

where ***<spec>*** is a timing specification of the form

<timing-check> = <limit>

Referencing a timing check by itself, without a qualifying pin name or clock level—**SETUP**, **HOLD**, **PW**—specifies *all* checks of that type. For example, in a **TIMING–CHECKS** block for a JKCF instance, **SETUP** specifies all setup checks; **SETUP.J, SETUP.K.**, **SETUP.NR**, and **SETUP.NS**. Values specified for qualified timing checks supercede those specified for unqualified checks.

As an example, the timing block:

```
part=FF1 type=DCF i=reset,set,clk,data o=q1 $
          timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD.D = 10; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

specifies:

1. All setups are 5 units.
2. All holds are 5 units, except hold from **D** which is 10.
3. All pulse-widths are 4, except clock-low pulse-width is 3.

Delay vs. loading curves, as well as **MINIMUM**, **TYPICAL**, and **MAXIMUM** delay sets can be used for specifying ***<limit>***. If delay curves are used, then the loading on the part's output is used to determine the timing check value. The syntax is identical to specifying local delays. For example:

```
timing–checks= $
BEGIN; $
SETUP = 25;20; $=maximum delay same as typical
HOLD = [4,2];[3,1];[5,3]; $
PW = (5,1)(7,3);4;[9,3]; $
end;
```

2.7.6.2 Controlling Spike Propagation

The Subsection *Combinational Timing Hazards* in Chapter 2.6 describes the **spike hazard**, a timing problem characterized by a sequence of changes at an element's inputs such that an element output begins responding to the first input state, but before its response time elapses, a second input change occurs, causing the output to return to its original value. This situation is illustrated in Figure 2.7-7 for a two-input AND gate.

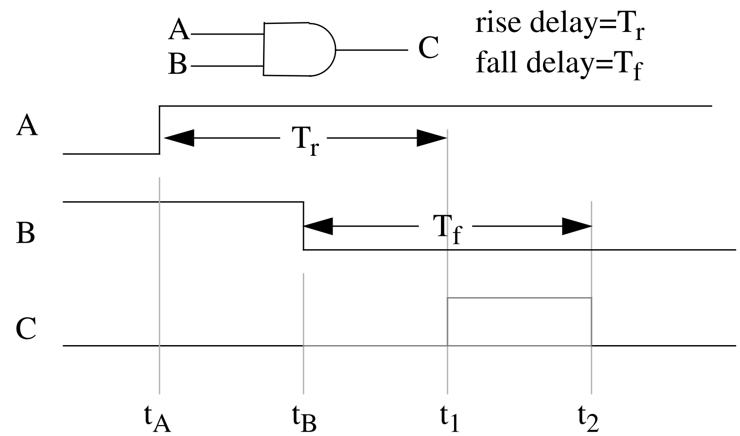


Figure 2.7-7 Spike Hazard At An AND Gate

In this figure, the rise at signal **A** would normally cause output **C** to rise at time

$t_1 = t_A + T_r$

However, at time $t_B < t_1$, signal **B** executes a transition to 0, which forces signal **C** to be 0 at, or before, time

$t_2 = t_B + T_f$.

What value or values should signal **C** be assigned in the interval between t_B and t_2?

Since propagation delays are inertial, i.e., associated with charging and discharging node capacitance, and since, by assumption, the time required to charge signal **C** to the logic-1 threshold is T_r, one possible answer to this question is to maintain **C** at a constant 0, since the peak voltage level it reaches at time t_B must be less than this threshold. This model, which ignores (filters out) the effects of transient input states whose duration is less than the output's response time, is called **inertial filtering**.

Alternatively, since simulation is usually performed to predict the circuit's operation in the real world, and since actual input arrival times, gate delays, and wiring delays will almost certainly differ from simulated values (the operation of circuits from separate production lots will also differ), it is very possible that an output pulse that *almost* happened during simulation will *actually* happen on some manufactured chips. Thus, another possible answer to the above question, based on a conservative approach, is to set the value of signal C to X within this interval, since actual operation may be indeterminate at the time of simulation. This model, which creates an X-pulse when a spike hazard is detected, and propagates this pulse to all fanouts, is called **spike propagation**.

SNL supports flexible per-node control on the degree of pessimism introduced into spike propagation. The SIMIC **`XPROPAGATE`** run command provides the same control at run time (see the Section *Enabling And Disabling X-Propagation* in Chapter 2.6).

Two parameters may be specified in any PART or TYPE statement to control spike propagation at each output:

1. **`filter`** – specifies a transient width *threshold*. Spike-producing transient input states that are narrower than this threshold will be inertially filtered, while transients at least as wide as the threshold will be propagated as X-pulses, if possible (the transient may still be filtered because of large differences between rise and fall delays; see below).
2. **`liberal`** – controls when to start the X-pulse. An X-pulse always ends at the time that the affected signal in guaranteed to have reached its known final value.

The **`filter`** threshold is expressed as percentage of the output's response time. Thus, in the above example, if this threshold is 10%, then the spike will be inertially filtered if the difference in arrival times of the events at **`A`** and **`B`**, $t_B - t_A$, is less than $0.1 \times T_r$. This value is specified as an integer between 0 and 100, inclusive, optionally followed by a percent (%) sign.

The value **`filter=0`** introduces the *greatest* pessimism, since any input transient is at least as wide as this threshold. The value **`filter=100`** introduces *no* pessimism, since all input transients are *inertially filtered* (this threshold corresponds to an input whose duration is the response time; but this is not a spike situation since the output has exactly enough time to respond).

The **`liberal`** value linearly controls the start of the X-pulse between the extremes of (a) the time of the input event causing the spike and (b) the time that the output would have responded to the first event. In the above example, these extremes are t_B and t_1. Adopting this notation to describe the general case, the X-pulse starts at time

$$t_{start} = t_B + \mathbf{<liberal/100)>} \times (t_1 - t_B)$$

where **<liberal/100>** is the **`liberal`** value expressed as a decimal fraction from 0 to 1. The **`liberal`** values is specified as an integer between 0 and 100, inclusive, optionally followed by a percent (%) sign.

The value **`liberal=0`** introduces the *greatest* pessimism, since it starts the X-pulse at t_B, the time that the spike hazard is detected. The value

liberal=100 introduces the *least* pessimism, since it starts the X-pulse at t_1.

Note that when an output's rise and fall delays differ sufficiently, its response to the second input event can be earlier than its response to the first event. Using the notation of the above example, t_2 may be earlier than t_1 if T_f is sufficiently small. Since the X-pulse always ends at t_2, and since it is possible to specify a **liberal** values that bring t_{start} close to t_1, it is therefore possible to specify a starting time for the X-pulse that is later than its ending time. If this situation occurs, the X-pulse is not generated, and the transient input state is effectively inertially filtered.

The SNL keywords that specify the **filter** parameter are **BUS-FILTER** (**BFILTER**) for busses and **OUTPUT-FILTER** (**OFILTER**) for outputs. Similarly, the SNL keywords that specify the **liberal** parameter are **BUS-LIBERAL** (**BLIBERAL**) for busses and **OUTPUT-LIBERAL** (**OLIBERAL**) for outputs.

Unspecified values for any of these keyword options are defaulted to 0, the most pessimistic settings. To globally disable spike propagation when verifying logical correctness rather than timing, use the

NO XPROPAGATE SPIKE:

command (see *Enabling And Disabling X-Propagation* in Chapter 2.6).

As an example, the PART statement

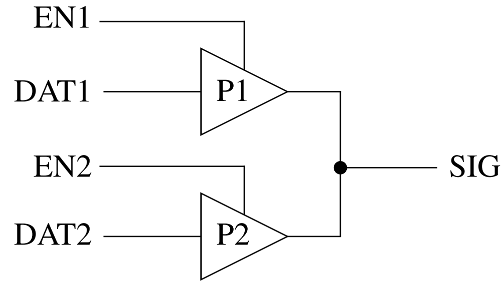
```
p=p15  t=xyz  i=a,b,c  o=d,e,f  $
   odel=dela,delb,delc ofilter=10%,,50  $
   oliberal=30,20%
```

assigns:

output **D** a filter value of 10% and a liberal value of 30%
output **E** a filter value of 0 and a liberal value of 20%
output **F** a filter value of 50% and a liberal value of 0

2.7.7 Wire-Ties

Wire-ties are created by assigning different element outputs or busses the same signal name. Figure 2.7-8 illustrates this situation for the outputs of two TPADN primitives having the identical output signal, **`sig`**. Wire-ties can also be created dynamically, when two differently-named signals are connected through an ON ideal switch (BTGN or BTGP).




```
p=p1  t=tpadn  i=en1,dat1  o=sig  odel=del1 $
       odecay=dec1
p=p2  t=tpadn  i=en2,dat2  o=sig  odel=del2 $
       odecay=dec2
```

Figure 2.7-8 Sample Wire-Tie

2.7.7.1 Wire-Tie Dominance

If, in Figure 2.7-8, only one of the TPADN elements drives **`sig`** (the other being disabled with its enable input at logical-0), or if both elements drive the same level, then the value at **`sig`** is well-defined. If the elements simultaneously drive different values, then a conflict exists, and the resulting value at **`sig`** depends on the specified wire-tie characteristics. SIMIC supports three types of wire-ties:

1. Wired-AND (**0-dominance**) – if any of the strongest components connected to the wire-tie is driving a logical-0, then the signal's value is 0
2. Wired-OR (**1-dominance**) – if any of the strongest components connected to the wire-tie is driving a logical-1, then the signal's value is 1.
3. CONFLICT (**X-dominance**) – if either (a) the state of at least one of the strongest drivers connected to the wire-tie is unknown, or (b) one or more of the strongest drivers is driving logical-0, while one or more of the remaining strongest drivers is simultaneously driving logical-1, then the signal's value is undefined (set to X).

A wire-tie's dominance can be specified in a PART or TYPE statement with the **`BUS-DOMINANCE`** (**`BDOM`**) keyword-field for busses or the **`OUTPUT-DOMINANCE`** (**`ODOM`**) keyword-field for outputs. These keywords associate in a one-to-one correspondence with the PART or TYPE statement's busses and outputs, respectively. The values expected for these keywords are the dominance categories **0**, **1**, or **X**.

For example, assume that a part has three outputs, each tied to other signals (but not to each other). If the wire-tie associated with the first output functions as a wired-AND, the wire-tie associated with the second output functions as a CONFLICT tie, and the wire-tie associated with the third output functions as a wired-OR, then **ODOM=0,,1** should be specified in the PART statement.

If the dominance is unspecified, the default wire-tie type is CONFLICT (**ODOM=X**). Thus, wire-tie dominance only requires specification at wire-tied signals whose mutual interaction functions as a wired-AND or wired-OR.

A wire-tie's dominance can be specified in *any* PART or TYPE statement associated with the common signal. If specified for more than one driver, the specifications must be identical.

For example, in the circuit of Figure 2.7-8, neither PART statement specifies a dominance for signal SIG, so by default, the wire-tie is X-dominant. Placing an output dominance specification in either PART statement, or in both, specifies the wire-tie's dominance, even if other drivers are connected to SIG. If placed in multiple statements, all dominance specifications must be identical.

2.7.7.2 Specifying Drive Strength

Most wire-tied configurations are designed to have at most one active driver at any one time (e.g., address and data lines between a CPU, ROM, and RAM). Sometimes, however, correct operation requires that a strong driver overpower a weaker one. For these situations, it is necessary to incorporate drive strength into driver models.

Drive strength can be specified in any PART or TYPE statement. This value can be one of the following reserved words:

POWER, DRIVING, RESISTIVE, FLOATING.

Any prefix of these words is valid.

High (pullup) and low (pulldown) drive strengths may be specified independently, or simultaneously if identical. The associated keywords are:

BUS-DRIVE (**BDRIVE**) – high and low drive strength for busses
BUS-LDRIVE (**BLDRIVE**) – low drive strength for busses
BUS-HDRIVE (**BHDRiVE**) – high drive strength for busses
OUTPUT-DRIVE (**ODRIVE**) – high and low drive strength for outputs
OUTPUT-LDRIVE (**OLDRIVE**) – low drive strength for outputs
OUTPUT-HDRIVE (**OHDRiVE**) – high drive strength for outputs

If a signal's high or low drive strength is unspecified, the default strength is DRIVING.

For example, the PART statement:

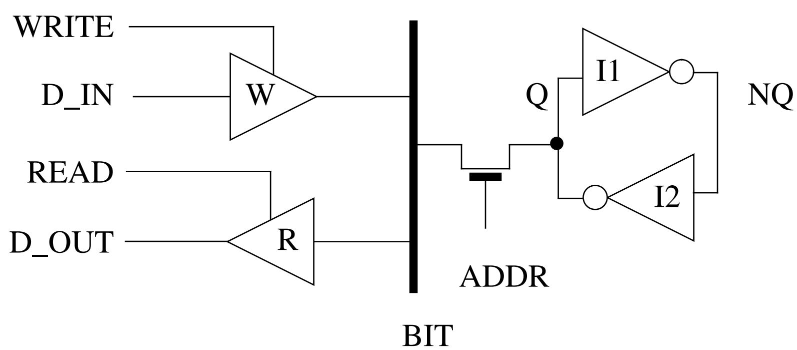
```
P=p22  T=xyz  I=a,b,c  B=d,e  O=f,g $
  BHDRIVE=,res  BLDRIVE=fl,pow ODRIVE=pow,res
```

assigns the following drive strengths:

bus **D** – high-drive=DRIVING, low-drive=FLOATING
bus **E** – high-drive=RESISTIVE, low-drive=POWER
output **F** – high-drive=low-drive=POWER
output **G** – high-drive=low-drive=RESISTIVE

Additionally, the BTGRN and BTGRP resistive bidirectional switches and the UTGRN and UTGRP resistive unidirectional switches can be assigned depths with the **SERIES–DEPTH (SDEPTH**) keyword-field. (The Subsection *Depths And Strengths* in Chapter 2.6 describes the correspondence between series depths and drive strengths.) The series depth value may be an integer between 1 and 32,767 inclusive. If unspecified, the default series-depth is 1.

Figure 2.7-9 illustrates a sample circuit that requires proper drive strength assignments in order to model its operation. The inverter loop represents a single memory cell inside a RAM. The **bit** line connects to all cells associated with a particular memory bit through series transistors, only one of which is ON at any time. The write amplifier, **w**, drives the **bit** line at POWER strength. When this value propagates through the series transistor, with a series-depth of 1, its strength is reduced to DRIVING. This is strong enough to overcome the RESISTIVE strength of inverter **i2**, so the value of **d_in** is written into the memory cell.




```
p=w  t=tpadn  i=write,d_in  o=bit  odrive=pow
p=r  t=tpadn  i=read,bit    o=d_out
p=x  t=btgrn  i=addr  b=bit,q  sdepth=1
p=i1 t=inv    i=q     o=nq
p=i2 t=inv    i=nq    o=q  odrive=resistive
```

Figure 2.7-9 Memory Cell Read/Write Circuit

2.7.8 Hierarchical Precedence of Electrical Attributes

SNL maintains a top-down hierarchical precedence for electrical attributes (delay, loading, decay, drive strength, IHIZ default) as well as for signal names. Electrical attributes specified at a higher level of the hierarchy override, or take precedence over, corresponding specifications at a lower level. Thus, any attribute specified in a PART statement overrides the corresponding value specified in the instantiated part's TYPE statement, which, in turn, overrides the corresponding attribute value specified in its constituent PART statements.

The following example illustrates hierarchical precedence. It defines a type block, **hier_demo**, that contains two **xor** macro instances (the XOR is a gate-level implementation of a 2-input exclusive-or). The sequential column numbers at the left are "source line" designations, that are referenced below, and are not part of the SNL description:

```
1.   !delay
2.   delay=del1  rise=3
3.   delay=del2  change=1

4.   !logical

5.   type=hier_demo  i=a,b,c,d  o=e,f  ilod=,,,10 $
         odel=,del1
6.     p=xor1  t=xor  i=a,b  o=e  ilod=7  olod=3
7.     p=xor2  t=xor  i=c,d  o=f  ohdrive=res $
         odel=del2

8.   type=xor  i=x,y  o=z  ilod=,8  olod=4
9.     p=n1  t=nand  i=x,y  ilod=1,1
10.    p=n2  t=or    i=x,y  ilod=2,2
11.    p=z   t=and   i=n1,n2  orise=5  ofall=6
```

The electrical attributes at the pins of an isolated **xor** type block are:

pin	loading	source line
x	3	9,10 (sum of pin loads)
y	8	8 (overrides 9,10)
z	4	8

pin	delay	source line
z	rise=5, fall=6	11

The electrical attributes at the pins of the **`hier_demo`** type block are:

pin	loading	source line
a	7	6 (overrides 9,10)
b	8	8 (overrides 9,10)
c	3	9,10
d	10	5 (overrides 8 which overrides 9,10)
e	3	6 (overrides 8)
f	4	8

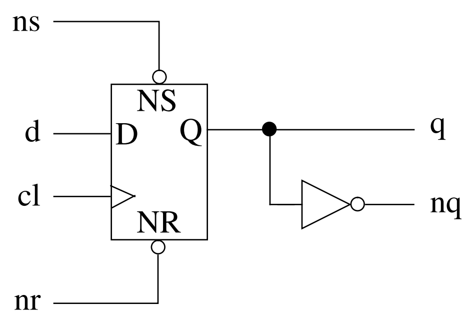
pin	delay	source line
e	rise=5 fall=6	11
f	del1	5 (overrides 7 which overrides 11)

Note that the delay at output **`f`** is **`del1`**, as specified in the type statement of **`hier_demo`**. This completely overrides the **`ODEL=del2`** specification in the PART statement for **`xor2`**, even though **`del1`** has no fall delay specified. This is because the missing fall delay specification in **`del1`** causes an implicit assignment of 0 for this delay, and the rise/fall delay pair of **`del1`** (3/0) overrides the rise/fall delay pair of **`del2`** (1/1).

2.7.9 Unused Bus and Output Pins

If a type block has a fixed number of pins, as defined in its TYPE statement, then it is an error if the number of inputs, outputs, or busses specified in an instantiating PART statement does not match those in the TYPE statement. Sometimes, however, output or bus pins of instantiated types are not used. In this case, the **`UNUSED`** reserved word should be specified as the “connection” for these pins; it is self-documenting, and it informs SIMIC that no error was made specifying connectivity.

For example, Figure 2.7-10 illustrates a macro, DFF, containing a DCF and in inverter to make the $\overline{\mathtt{q}}$ output available. Also shown are two part statements (within another type block) that instantiate this macro. The second instance does not use the **`q`** output of the macro. Note that because of the **`UNUSED`** entry, the **`q`** signal of instance **`ff2`** has no user-assigned name. It can still, however, be examined during simulation, if necessary. An **`UNUSED`** pin is “pushed into” the macro, for this macro instance. Thus, the **`q`** signal in **`ff2`** can be referenced as **`ff2.q`**.




```
type=dff i=nr,ns,cl,d  o=q,nq
   p=q  t=dcf  i=nr,ns,cl,d  o=q
   p=nq t=inv  i=q

  ....................................
c= these part statements are in a different
c= type block
p=ff1  t=dff  i=nr1,ns1,cl1,d1  o=q1,nq1
p=ff2  t=dff  i=nr2,ns2,cl2,d2  o=unused,nq2
```

Figure 2.7-10 Example of Unused Instance Pin

2.7.10 Specifying Level of Abstraction

Sometimes, when a subcircuit requires optimization or when multiple designs of the subcircuit are being investigated, multiple descriptions of the subcircuit will be created and tested. It is convenient to assign all variants the same type name, since it may be instantiated many times; using different names would require editing the netlist each time another variant is tested.

SIMIC searches for an instantiated **type**'s definition in the order listed here, unless the instantiating PART statement explicitly specifies its **composition**.

Any component instantiated in a PART statement will have one of the following levels of abstraction:

```
MACRO, BOOLEAN, BEHAVIORAL, PRIMITIVE.
```

It is an error to assign the same type name to multiple type blocks at the same level of abstraction. For example, every macro's type block must have a unique name. If multiple type blocks at the same level of abstraction have the same name, SIMIC will accept the first definition found, and issue a warning message that multiple type blocks have identical names. The solution is to place the variant definitions in separate files, and select the appropriate file with the **GET FILE** keyword option.

In contrast, identically-named variants defined at different levels of abstraction *can* be placed in the same file. For example, one definition of a full-adder could be a macro, and another could be, say, a BOOLEAN type. The **COMPOSITION** keyword-field is used to specify which model (level of abstraction) is referenced by each instantiating PART statement.

If the **COMPOSITION** keyword is not specified, the SIMIC compiler searches for definitions in the order listed above. Thus, in the above situation, SIMIC would always instantiate the MACRO full-adder definition. To instantiate the BOOLEAN definition:

```
p=fad  t=full-adder  i=a1,b1,c1  o=s1,co1  $
     composition=boolean
```

SIMIC also issues a warning message when type blocks at different levels of abstraction have the same name. For example (although not good practice), it is possible to define a MACRO or BOOLEAN having the same name as a SIMIC primitive, which would always be instantiated instead of the primitive unless the PART statement contains **COMPOSITION=PRIMITIVE**. This could lead to considerable confusion without the message.

The **composition** keyword should also be used for a type block containing only the **type** statement of a BOOLEAN whose equations are supplied at run-time.

Although unrelated, the **COMPOSITION** keyword serves another function. It is an error to define a type block containing only a TYPE statement unless it is a BOOLEAN type block. In this case, the TYPE statement would contain a BOOLEAN equation block identifying it as such. Sometimes, however, when experimentation is necessary, it may be desirable to omit the equations from the SNL description and specify them at run time, via the **CLAMP** command. In this case the keyword-field **COMPOSITION=BOOLEAN** should be placed in the TYPE statement to identify it as a BOOLEAN type in place of the missing equations.

2.7.11 Physical Size Metrics

SIMIC supports three metrics that can provide assistance in obtaining estimates of the main type's physical size. These metrics are:

- **`PADS`** – the number of physical pads for the cell
- **`TRANSISTORS`** (**`TRANS`**) – the number of transistors in a cell
- **`WIDTH`** (**`W`**) – the width of a cell

These keyword-fields may be placed in any PART or TYPE statement. Their associated values are integers:

PADS=<*integer)*
TRANS=<*integer*>
WIDTH=<*integer*>

SIMIC makes no use of these numbers other than to total them during compilation, whenever a PART or TYPE statement containing the metrics is encountered, and report their respective sums.

In a standard cell methodology, the metrics should be placed in the TYPE statement for each cell, and the instantiating PART statements should not contain them. The resulting sums reported by SIMIC would then represent the total physical contributions of all instantiated cells.

In a custom design methodology, the cell TYPE statements might contain metrics for the smallest possible cell, and instantiating PART statements might contain the differences between these values and the actual instance values.

For example, if the TYPE statement of the **`full-adder`** in Figure 2.7-2 is changed to include size metrics:

```
type=full-adder  i=ia,ib,c-in  o=sum,c-out $
          trans=26 width=50
```

the messages SIMIC issues in response to the **`GET`** command for compiling the **`four-bit`** adder would be:

```
Main Get Network : FOUR-BIT
GET completed, Circuit totals:  Parts = 21; Signals = 32
   Inputs = 9; Busses = 0; Outputs = 6
Physical Totals:
   Width = 200
   Transistors = 104
```

Chapter 2.8 Tester Interface

2.8.1 Introduction

Present day test equipment offers flexibility beyond that found only a few years ago. However, there are common resources associated with all test equipment, such as programmable master clock periods, timing generators that shape device inputs and strobes that monitor device outputs. SIMIC's tester-oriented stimulus and strobe definition language emulates the tester environment, allowing test programs to be described concisely and then be debugged during simulation. This has the distinct advantage of minimizing the amount of precious production tester time required for debugging. SIMIC also provides special information about bidirectional pads to insure accurate, non-destructive tests. Finally, since circuit faults are made visible by strobes placed exactly where they will occur on the tester, this mode of simulation allows the most accurate prediction of fault coverage by the SIMIC fault simulator.

2.8.2 Tester Emulation Mode

2.8.2.1 Defining Master Test Period

Defining a non-zero period automatically causes SIMIC to enter tester emulation mode.

The master period specifies a time interval corresponding to a tester cycle. The start of each interval begins a new test. The default test period is defined with the **PERIOD** (**PE**) keyword option of the **DEFINE** (**DE**) command:

```
DEFINE PERIOD=<n>
```

where <***n***> is the number of time-units in each test period. Defining a period automatically switches SIMIC into tester emulation mode, and it will remain in tester emulation mode until the master test period is explicitly removed. This is accomplished by setting the value to 0 with the command:

```
DEFINE PERIOD=0
```

The size of the period can be changed during simulation with the **APPLY** (**AP**) **PERIOD** run command and its **BEGIN** (**BE**) option. Its format is:

```
APPLY PERIOD=<n> BEGIN=<m>
```

where <***n***> is the size of the new test period in time-units, and <***m***> is the test number at which this period becomes effective. Note that the **DEFINE PERIOD** command is functionally equivalent to the **APPLY PERIOD** command with a **BEGIN** value of 1.

2.8.2.2 Defining Drive Values

The **drive** values, which the tester would apply to primary inputs and primary bidirectional busses functioning as inputs, are specified by defining **patterns**. Syntactically, tester emulation patterns are identical to the simulate-till-stable patterns described in Chapter 2.3. However, when the global period has been set to a non-zero value (with the **`DEFINE PERIOD`** command), SIMIC successively applies the next pattern at the beginning of the next test period, rather than at the time the circuit state stabilizes.

2.8.2.3 Defining Timing Generators

Timing generators are essentially timing **masks** used in conjunction with the current drive value. Two types of masks can be specified; a **drive** mask, which controls logic levels driven within the test period, and an **enable** mask, which controls when and whether to tristate drives within the test period. All masks are defined by event times, called **marks**, that are referenced to the start of the test period.

One of the following four different drive mask types can be used for each defined timing generator:

1. Non-Return-to-Zero (**`NRZ`**). This mask allows rise and/or fall transitions to be independently skewed from the beginning of the test period.
2. Return-to-Zero (**`RZ`**). This mask is used to define the shape of a positive pulse, generated whenever the drive value is a logical-1 for that period. When the drive is logical-0, the signal remains logical-0 throughout the entire test period[1].
3. Return-to-One (**`RO`**). This mask is used to define the shape of a negative pulse, generated whenever the drive value is a logical-0 for that period. When the drive is logical-1, the signal remains logical-1 throughout the entire test period[1].
4. Return-to-Complement (**`RC`**). This mask is commonly used to test setup and hold requirements. It specifies the shape of a negative pulse, when the drive is logical-0, or a positive pulse, when the drive is logical-1, that is preceded and followed by the drive's complement.

Additionally, one of the following three enable mask types can be used for each defined timing generator to control when and whether the drive is tristated (Z).

1. No-Envelope (**`NE`**). This mask allows drive-to-tristate and tristate-to-driving transitions to be independently skewed from the beginning of the test period.

1. This may not always be true if the timing generator definition (as defined by its mark locations) extends over multiple periods (**pulse-extended** periods, as described below).

2. Return-to-Drive (**`RD`**). This mask is used to shape the tristating window when the driving pattern is a Z.
3. Return-to-Float (**`RF`**). This mask is used to shape the driving window when the driving pattern is not a Z.

It is important to note that SIMIC retains the previous drive value during periods where the drive is specified as Z. This history determines the behavior of the signal when enable mask skews and drive mask skews are not identical. To understand this behavior, view SIMIC as always producing a drive value consistent with the drive format mask and the current (or retained) drive value, which is then connected (or disconnected) as specified by the enable format mask.

The combination of drive and enable masks constitute a SIMIC **time-set**. The time-set is specified with the **`DEFINE`** command. One format for this definition is:

`DEFINE T`*<name>*`.`*<dformat>*`=`*<dmarks>*

where ***<name>*** is a user-defined name for the time-set, ***<dformat>*** is one of the drive mask types described above, and ***<dmarks>*** is a list of marks, or timing values, whose order and number depends on the ***<dformat>*** selected. In this format, no enable mask was specified.

If not specified, then the enable mask defaults to **`NE`** with 0 skewing, with one exception: if the **`NRZ`** driving format mask is used and the rise and fall skews are identical, then the default is **`NE`** with skewing that matches the rise and fall skews.

In order to specify the enable mask the following format is used:

`DEFINE T`*<name>*`.`*<dformat>*`.`*<eformat>*`=`*<dmarks>*`;`*<emarks>*

Note the addition of ***<eformat>***, which selects one of the enable formats described above, and ***<emarks>*** which is a list of timing values, whose order and number depends on the selected enable format.

Table 2.8-1 describes the drive mask formats and their associated timing marks.

SIMIC does not require mark specifications to be restricted to a single period. A period is called **pulse-extended** if it continues an active time-set spanning multiple periods. When a new time-set is applied, it is possible for its marks to overlap with the marks of the previous time-set, if the latter's period was pulse-extended. In this situation, SIMIC merges the specifications by following the individual timing mark actions, in the order that they occur. This transient situation ends with the last mark of the previous time-set.

Drive Mask Format	Order And Number Of Marks	Description
NRZ	O, Z	O specifies the time to go to logical-1, when the drive value is logical-1. Z specifies the time to go to logical-0, if the drive value is logical-0. If Z is omitted, then Z is set to the value specified for O.
RZ	O, Z1, Z2	O specifies the time to go to logical-1, when the drive value is logical-1. Z1 specifies the time to go to logical-0, regardless of the drive value. Z2 specifies the time to go to logical-0, only when the time-set is first applied (see explanation in text). If omitted, then Z2 is set to 0. In order for the definition to be valid, the magnitudes of the marks must be ordered as follows: Z2 < O < Z1
RO	Z, O1, O2	Z specifies the time to go to logical-0, when the drive value is logical-0. O1 specifies the time to go to logical-1, regardless of the drive value. O2 specifies the time to go to logical-1, only when the time-set is first applied (see explanation in text). If omitted, then O2 is set to 0. In order for the definition to be valid, the magnitudes of the marks must be ordered as follows: O2 < Z < O1
RC	T, C1, C2	T specifies the time to go to the drive value. C1 specifies the time to return to the complement of the drive value. C2 specifies the time to go to the complement of the drive value, prior to T. If C2 is omitted, then C2 is set to 0. In order for the definition to be valid, the magnitudes of the marks must be ordered as follows: C2 < T < C1

Table 2.8-1 Drive Mask Formats

Enable Mask Format	Order And Number Of Marks	Description
NE	F, D	F specifies the time to disable the drive, when the pattern value is ‘Z’. D specifies the time to enable the drive, when the pattern value is not a ‘Z’.
RD	F, D	F specifies the time to disable the drive, when the pattern value is ‘Z’. D specifies the time to enable the drive, when the pattern value is ‘Z’. In order for the definition to be valid: $F < D$
RF	D, F	D specifies the time to enable the drive, when the pattern value is not ‘Z’. F specifies the time to disable the drive, when the pattern value is not ‘Z’. In order for the definition to be valid: $D < F$

Table 2.8-2 Enable Mask Formats

Unfortunately, testers differ in whether they support pulse-extended periods and in how they transition between old and new time-sets. The timing marks Z2 in the RZ drive format and O2 in the RO drive format are only relevant during this transition period, when the new time-set is first applied. They allow injection of a mark forcing the new time-set to become active (e.g., for the RZ format, the signal is forced to 0 at time Z2, regardless of the previous timing-set’s definition). If unspecified, Z2 and O2 default to 0, causing a new RZ or RO time-set to become active at the beginning of the period.

Table 2.8-2 describes the enable mask formats and their associated timing marks.

Any timing mark can be <u>suppressed</u> by placing an asterisk (*) as a substitute for the value. For example, to prevent an **NRZ** timing generator from tristating, even when the pattern value is ‘Z’, an asterisk should be placed in the ‘F’ value of the **NE** enable format specification, i.e:

```
Define tnofloat.nrz.ne=500,500;*,0
```

Care should be taken to limit time-set definitions to the capabilities of the target tester.

The number of definable time-sets, active time-sets, number and type of time-set switches, mark suppression, number of periods for which a time-set can be active, and other properties may be restricted by the target tester’s capabilities. Therefore, timing-set definitions should be limited to the capabilities of the target tester. For example, while SIMIC supports definition of an unlimited number of time-sets whose active interval can span an arbitrary number of (pulse-extended) test periods, testers contain fixed limits on these resources.

Figure 2.8-1 illustrates the effects of the drive format mask selection:

```
Define period=1000
Define tnrz.nrz=300
Define trz.rz=300,700
Define tro.ro=300,700
Define trc.rc=300,700
```

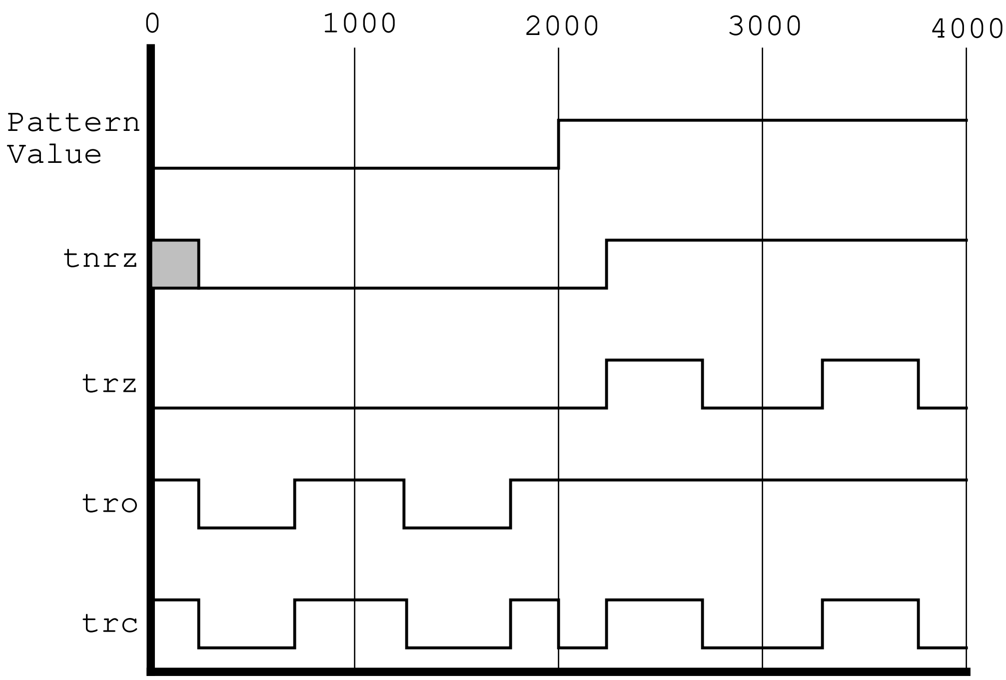


Figure 2.8-1 Effect Of Sample Drive Format Masks

Note that the **`NRZ`** format starts in an unknown (X) state in the first period.

Figure 2.8-2 illustrates the effects of the enable mask formats. Note: the shaded sections are disabled (Z) drivers:

```
Define period=1000
Define tne.nrz.ne=0,0;500,500
Define trd.nrz.rd=0,0;300,700
Define trf.nrz.rf=0,0;300,700
```

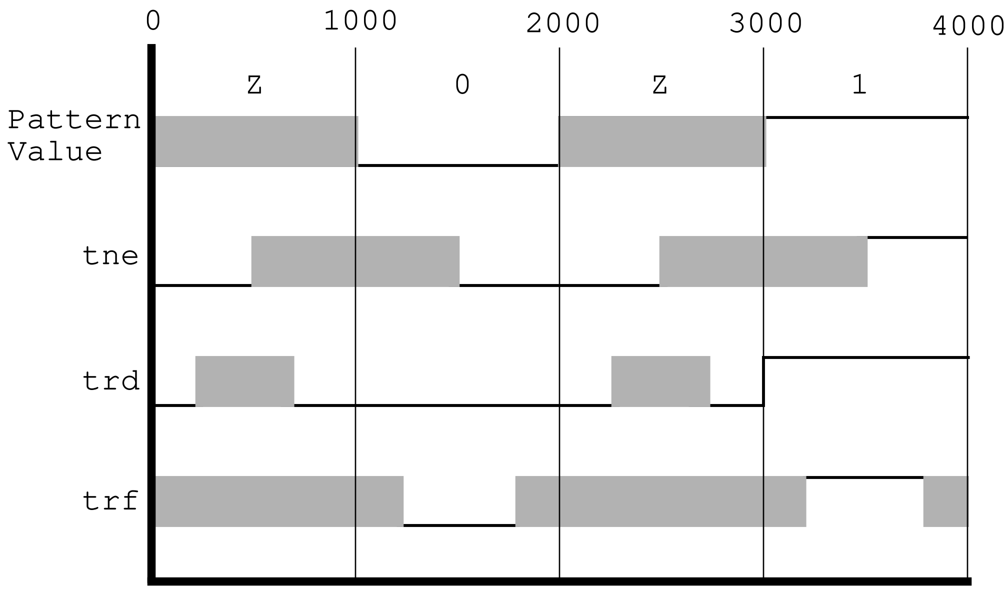


Figure 2.8-2 Effect Of Sample Enable Format Masks

2.8.2.4 Assigning Time-Sets to Input and Bidirectional Pads

Once a time-set has been defined as described above, it is attached to primary pads with the **`APPLY`** (**`AP`**) **`TIMING`** (**`TI`**) run command and the **`LIST`** (**`LI`**) keyword option. The syntax of this command is:

`APPLY TIMING=`<*time-set*> `LIST=`<*signal list*>

where ***<time-set>*** is the name of a defined time-set (e.g., **`tne`**), and ***<signal list>*** is the list of primary signals to be assigned the specified time-set. If a primary input/bidirectional pad has not been assigned a timing generator, then the default generator (**`TDEFAULT.NRZ.NE=0,0;0,0`**) will be applied.

Time-sets may be changed at any time during simulation with the **`APPLY TIMING`** command with the **`BEGIN`** (**`BE`**) keyword option. The form of this command is:

`APPLY TIMING=`<*time-set*> `LIST=`<*signal list*> `BEGIN=`<*m*>

where ***<m>*** is the test number where this time-set is to be applied.

2.8.2.5 Defining Strobes

SIMIC supports two different types of strobes. The first is a point (or edge) strobe (**`SP`**) and the second is a window strobe (**`SW`**). In SIMIC, the point strobe can be viewed as a window strobe of zero width. The strobed signal must maintain a constant value within the active window of the strobe, otherwise a **strobe error** will result. By default, a strobe warning will be generated for each strobe error. This warning may be disabled with the command:

```
NO WARN STROBE:
```

and re-enabled with the command:

```
WARN STROBE:
```

In addition, a breakpoint can be set if a strobe error occurs with the command:

```
BREAK STROBE:
```

This breakpoint can be disabled with the command:

```
NO BREAK STROBE:
```

The format for defining a strobe is:

```
DEFINE S<name>.<type>=<position>
```

where ***<name>*** is an user defined name for the strobe, ***<type>*** is either **`SP`** (for a point strobe), or **`SW`** (for a window strobe), and ***<position>*** is a single value for **`SP`**, indicating the time to fire the strobe, or two values for **`SW`**, indicating the time to start and the time to stop the window strobe, respectively. All times are in time-units, relative to the start of the period. For example, the following command defines a window strobe, whose window begins at time 600 and ends at time 700:

```
DEFINE SDEMO1.SW=600,700
```

To define a point strobe at time 800:

```
DEFINE SDEMO2.SP=800
```

2.8.2.6 Assigning Strobes to Outputs and Bidirectional Pads

Strobes are assigned to primary signals in the same fashion as time-sets, described above, with the **`APPLY TIMING`** run command. This format is:

```
APPLY TIMING=<strobe> LIST=<signal list>
```

where ***<strobe>*** is the name of the strobe (e.g., **`SDEMO1`**), and ***<signal list>*** is the list of primary output/bidirectional signals to assign the specified strobe.

Strobes may be changed at any time during simulation with the **`APPLY`**

`TIMING` command with the **`BEGIN`** keyword option. The form of this command is:

APPLY TIMING=<*strobe*> LIST=<*signal list*> BEGIN=<*m*>

where ***<m>*** is the test number where this time-set is to be applied.

For any output that does not have a strobe assigned, a default point strobe will be placed one time unit prior to the end of the test period.

2.8.3 Test Program Output

2.8.3.1 Introduction

SIMIC has a special file format for interfacing with testers, called the **tester interface file**. This file has the default extension of **`tgn`**, and is created by issuing the **`TGEN`** (**`TG`**) command and the **`FILE`** (**`FI`**) keyword option:

TGEN FILE:

or

TGEN FILE=<*filename*>

prior to simulation, where ***<filename>*** is a user-supplied name for the file. This file is supported for all three stimulus modes.

Since testing involves the final design, the simulation that generates the tester interface file should utilize the most accurate estimates of propagation delays, wiring delays, and timing checks. No degree of unwarranted optimism should be introduced in this simulation. To insure that timing checks and combinational hazard analysis are not defeated or made more lenient, the tester interface file will not be generated if any run command relaxes or restricts X-propagation for timing checks and combinational hazards.

2.8.3.2 Tester Interface File Contents

The tester interface file will contain the following sections:

1. A time/date/version stamp - This consists of a single line **`REMARK`** that contains the time, date and SIMIC version that created the file.
2. A target specification - The intended tester is specified with the **`TGEN`** command and the **`TARGET`** (**`TA`**) keyword option:

   TGEN TARGET=<*name*>

   where ***<name>*** is the tester's name. If unspecified, then the name defaults to "???".

3. A Channel section - This section contains, for each primary signal:
   - The signal's name.
   - A character (I, O, or B) specifying that the signal is a primary input, primary output, or primary bus, respectively.
   - A pin field. This field is always set to unknown (-) by SIMIC.
   - A column field. This field locates the position in the output section (see (7) below) that contains the signal's value. If this is a bidirectional signal, then the value in this position corresponds to the applied pattern value of this signal.
   - A channel field. This field is set unknown by SIMIC, either with no entry (for input or output signals) or a hyphen (-) for bidirectional signals. The value of this field specifies the tester channel to assign to this signal.
   - A second column field. This field is only present if the signal is bidirectional. It specifies the position in the output section (see (7) below) that contains the value of the wire-tied signal (when the **DISCONNECT** option is disabled), or the value of the wire-tied signal, excluding the primary drive value (when the **DISCONNECT** option is enabled). The **DISCONNECT** (**DI**) option is enabled by default; to disable this feature, use the run command:

     ```
     NO TGEN DISCONNECT:
     ```

     and to re-enable it:

     ```
     TGEN DISCONNECT:
     ```

     or

     ```
     TGEN DISCONNECT=ALL
     ```

     The first option will disconnect the primary drive value, except when there is a wire-tie conflict (value will be X). The second option, **ALL**, will disconnect the primary drive value for all situations.
4. A Run section - This section begins with an **!RUN** statement. It only exists if tester emulation mode is in effect. The contents of this section define:
   - The default period (specified by the **DEFINE PERIOD** command).
   - The correspondence between SIMIC time-units and real-time. This would have been specified in the **!DELAY** section of the network description. If it was not specified, then the value, "???", is displayed, to indicate that this value is not known.
   - The time-set, and strobe definitions (**DEFINE T**<*name*> and **DEFINE S**<*name*> Test commands respectively).
   - The time-set and/or strobe to signal assignments (**APPLY TIMING** Test command).

5. A simulations options header - This consists of a number of **REMARK** lines that describe the options selected for simulation.
6. A signal name header - This consists of a number of **COMMENT** lines that contain the names of the signals to be reported, arranged in columns according to the positional placement of the primary signals in the output section.
7. A test output section - This section begins with an **!TGEN** statement. It contains a record for each test. For simulate-till-stable mode, a test output is generated each time the circuit becomes stable., for waveforms, a test output is generated whenever the circuit becomes stable, or a primary input changes while the circuit is unstable, and for tester emulation mode, a test output is generated whenever a new period is entered. Each record contains the word **TEST**, followed by the current test number, followed by the simulation test output values, in groups of 5 up to 50 values per line. The "spillover" lines will not contain the **TEST** or test number field.

   SIMIC automatically compresses identical test output. This can occur for two reasons. First, in simulate until stable mode, consecutive input patterns are identical. Second, in test emulation mode, consecutive input patterns are identical, and the strobed results do not change. This can occur in sequential logic, where several clock cycles are needed to propagate values to the output. These compressed pattern records will start with the word **DO**, followed by a number representing the number of times to repeat this test, followed by an open parenthesis, followed by the test output in the same format as the **TEST** records, followed by a close parenthesis.

   Values in this table will be only (0, 1, X or Z), since strength is not important to the test equipment. However, in switch level circuits, there can be very weak "sneak" paths that the tester should treat as high impedance. The threshold depth at which a path should be considered weak, and a 'Z' should be output to the tester interface file, is specified with the **HIZ** option of the **TGEN** command. For example:

   ```
   TGEN HIZ=20000
   ```

   specifies that any paths that have a resultant depth of 20000 or more will be represented as 'Z'.
8. Remark statements - besides the above mentioned **REMARK** output, SIMIC will place any **REMARK**s in the run command statements, issued after the **TGEN FILE** command into the file. This allows designers to include commentary to the test engineers directly in this tester interface file. Additionally, SIMIC places SNL remarks read from the backannotation file into this section of the file; thus, net loading information will be available if timing problems are discovered.

Figure 2.8-3 illustrates a sample circuit description and run file that defines timing generators and strobes and specifies that a tester interface file be generated.

Figure 2.8-4 illustrates the resulting tester interface file:

```
c= Network Description File
!f p= t= i= o=
type=tgntst         a         oa,ob     b=b
%declare hex=a[0:1]
oa        and       a[0]
ob        and       a[1]
b         tpad      one,a[0]
b1        tpad      one,a[1]  b
pu        tpad      one,one   a[1] odrive=res

c= Run File
define file=tgntst
get type=tgntst
c= ** Define and apply patterns
define p1.3 = 000 01x 010 000 zzz 000
apply pattern=p1
c= ** Define time-sets and strobes
define t1.ro=10,20
define t2.nrz=5
define s1.sp=29
define pe=30
c= ** Apply time-sets and strobes
apply timing=t1 list=a[0]
apply timing=t2 list=a[1]
apply timing=s1 list=&busses,&outputs
c= ** enable tester interface file
c= ** and specify target tester
tgen file: target=testamatic
simulate
quit
```

Figure 2.8-3 Sample Circuit Description And Run File To Illustrate Tester Interface File

```
Remark= 'Tgen' by SIMIC Version 1.00.00 on Wed Oct 2 09:21:06 1991

Define Target=TESTAMATIC
!Channel
C=  Name                  IO   Pin   Col   Chan   Col
    A[0]                  I     -      1
    A[1]                  I     -      2
    B                     B     -      3     -      4
    OA                    O     -      5
    OB                    O     -      6
!Run
Define Period=30
Define Time-Units=???
Define S1.SP=29
Define T1.RO.NE=10,20;0,0
Define T2.NRZ.NE=5,5;5,5
Apply Timing=T1 List=A[0]
Apply Timing=T2 List=A[1]
Apply Strobe=S1

Remark= Options: (Fault Free Simulation)
Remark=   Tester Stimuli, Near Filter, Spike Propagation
Remark=   Stable After Decay, Dynamic Delay

C=           A..BO O
C=           [AB A B
C=           0[
C=           ]1
C=            ]

!Tgen
TEST     1   000X1 0
TEST     2   01X11 1
TEST     3   010X1 1
TEST     4   000X1 0
TEST     5   ZZZXX 1
TEST     6   000X1 0
```

Figure 2.8-4 Tester Interface File Generated For Run Of Figure 2.8-3

Chapter 2.9 The History Files

2.9.1 Description

If requested, SIMIC can save signal event history information in binary files. The purpose of these files is to interface to simulation display and analysis programs. Two files are created: the general history file and the sequential history file. Both files store information in a compressed format to significantly reduce their size.

Access routines can be provided to allow easy generation of other formats from the history file format. Contact Genashor to find out if a particular format conversion program is already available or can be provided.

2.9.1.1 The General History File

The default extension for this file is **`hig`**. This file contains:

1. Circuit statistics. This is used to check the integrity between this and the sequential history file and to provide information on the signals that are included in the history.
2. Circuit state dump information. These are snapshots of the state of the network at specified intervals. The snapshots are used to provide a fast way of "jumping" from one time to another. It also provides a method of "resyncing" the state of the network when history information is suppressed (with the **`PRANGE`** keyword option).

2.9.1.2 The Sequential History File

The default extension for this file is **`his`**. This file contains signal, time, and test information for simulation events, as well as indications of when history information is suppressed during the simulation.

2.9.2 Enabling History File Generation

History file generation is enabled by selecting the signals to trace with the **`HISTORY`** (**`HI`**) command's **`LIST`** (**`LI`**) keyword option:

    HISTORY LIST:

to select all signals not masked by the **`STRING`** option, or:

`HISTORY LIST=`<*list of signals*>

to select the signals specified in ***<list of signals>*** not masked by the **`STRING`** option.

The **`STRING`** masking options are described in the Section *Name-Based*

Filtering this chapter.

The **NO** prefix may be used with the **LIST** keyword to remove signals. For example, the sequence of commands:

```
HISTORY LIST:
NO HISTORY LIST=abc,def
```

would cause all signals (not filtered by the **STRING** option) to be traced in the History file, except signals **abc** and **def**.

2.9.3 Restricting History Output To A Specified Interval

HISTORY output can be restricted to a specified test (patterns) or time (waveforms) with the **PRANGE** option of the **HISTORY** command. See the Section *Restricting Simulation Options To A Specified Simulation Interval* in Chapter 2.6.

2.9.4 Specifying a Dump Interval

Once enabled, dumps to the General History file are performed every 100 test steps (by default). This default can be changed with the **PSTEP** (**PS**) keyword option:

```
HISTORY PSTEP=<n>
```

where ***<n>*** is the test interval to perform the dumps. The dumps can be disabled with the command:

```
NO HISTORY PSTEP:
```

2.9.5 Specifying the History File Names

By default, SIMIC uses the default file name to construct the history file names. This default can be overridden with the **FILE** (**FI**) keyword option:

```
HISTORY FILE=<file name>
```

The default file name can be re-specified with:

```
HISTORY FILE:
```

It is important to note that the default extension should *not* be overridden, since the general and sequential history files must have unique names.

2.9.6 Name-Based Filtering

2.9.6.1 Overview

While the **LIST** keyword provides flexibility in selecting and excluding signals to be traced in the History file, this method can be inconvenient for situations where many signals distributed throughout the circuit should be excluded. One such situation arises in standard cell methodologies, where the values of certain (or all) signals internal to the library cells either need not, or should not, be saved for display.

Name-based filtering is a method of systematically **excluding** signals from the History file. Thus, it does **not** operate on **LIST** specifications when the **NO** command prefix is used.

SIMIC supports a methodology of automatically filtering signals based either on their names, or on the names of the parts that generate them. This filtering mechanism does not by itself select new signals or reject previously selected signals; rather it acts as a screen for signals specified in future **LIST** keyword-fields. The filter screen is *not* active with the **NO** prefix.

Two basic types of name-based filters are currently supported:

1. Part name filtering. A signal is filtered if the lowest-level component of the generating part's hierarchical name is numeric or alphanumeric (that is, non-numeric).
2. Signal name filtering. A signal is filtered if its hierarchical name matches any predefined filtering pattern (template).

Obviously, naming conventions for parts and signals internal to library cells must be compatible with these two filter types, in order to fully take advantage of SIMIC's automatic name-based filtering.

By default, no name-based filtering is performed for the **HISTORY** command.

2.9.6.2 Filtering Based On Part Names

If the parts that generate internal cell signals are all assigned, say, numeric names, then part name filtering can be used to automatically exclude these signals.

For example, suppose that a cell library has a **full-adder** cell, and that this cell is defined as shown in Figure 2.9-1. Signals **and1**, **and2**, and **and3** are all internal to the cell, and their generating parts have been assigned instance names whose characters are strictly numerics (0-9). The remaining signals, **sum** and **carry-out**, are not internal signals, and their generating part names are not numeric (although some characters in these names may be numerics, e.g., **or1**).

```
c= Full-Adder cell
t=full-adder i=a,b,carry-in o=sum,carry-out
   p=xor  t=exor  i=a,b,carry-in    o=sum
   p=1    t=and   i=a,carry-in      o=and1
   p=2    t=and   i=b,carry-in      o=and2
   p=3    t=and   i=a,b             o=and3
   p=or1  t=or    i=and1,and2,and3  o=carry-out
```

Figure 2.9-1 Full-Adder Cell For Numeric Filtering

A filter for the internal signals is set up with the **NO** prefix and the **STRING** (**STRI**) keyword option (the **NO** prefix specifies "do *not* include names of the specified type"):

```
NO HISTORY STRING=&NUMERIC
```

Any prefix of the word **NUMERIC** is valid (e.g., **&N**, **&NU**, etc.).

The leading "**&**" specifies filtering based on **part names.** Omitting it causes filtering based on **signal names**.

Caution: SIMIC will also accept **NUMERIC**, without the leading **&**. This would be another valid name-based filter whose effects are entirely different from **&NUMERIC**. See *Filtering Based On Signal Names* below.

Filtering will occur when subsequent signals are specified with the **HISTORY** command's **LIST** keyword.

Note that the hierarchical part name of the instantiated full-adder need not be numeric; only the lowest component of the part name is relevant. For example, suppose that two **full-adder** instances in the circuit are:

```
p=a  t=full-adder  i=a1,b1,cin1  o=s1,cout1
p=b  t=full-adder  i=a2,b2,cin2  o=s2,cout2
```

The names of the parts generating the internal signals will be **a.1**, **a.2**, **a.3**, **b.1**, **b.2**, and **b.3**; the lowest components of these hierarchical names are all numeric. If the command:

```
HISTORY LIST:
```

is issued after the above command, then the output signals of these six parts will not be present in the History file.

A filter can be removed (disabled) by re-specifying it without the **NO** prefix:

```
HISTORY STRING=&NUMERIC
```

Filters can be introduced and removed as necessary. For example, the run command sequence:

```
NO HISTORY STRING=&NUMERIC
HISTORY LIST=a.()
HISTORY STRING=&NUMERIC
HISTORY LIST=b.()
```

causes signals **a.and1**, **a.and2**, and **a.and3** to be absent from the History file, and signals **b.and1**, **b.and2**, **b.and3** to be included (a specification such as **a.()** is a *wildcard*—it means "all signals whose names begin with **a.**". See the Subsection *Signal Specification Options* in Chapter 2.4 for a full description of signal name specification).

In addition to numeric part name filtering, SIMIC also supports filtering of signals whose generating elements have alphanumeric (non-numeric) part names. This filter is introduced with the command:

```
NO HISTORY STRING=&ALPHANUMERIC
```

Any prefix of the word **ALPHANUMERIC** is valid (e.g., **&A**, **&AL**, etc.).

This filter is removed with the command:

```
HISTORY STRING=&ALPHANUMERIC
```

To use this filter for the full-adder cell, the part naming convention would be reversed; the parts generating signals **and1**, **and2**, and **and3**, would be assigned non-numeric names, while the parts generating **sum** and **carry-out** would be assigned numeric names.

The **&NUMERIC** and **&ALPHANUMERIC** filters are complemetary; specifying both would filter *all* subsequently-specified signals. SIMIC issues a warning message if both filters are activated, and will perform the filtering.

2.9.6.3 Filtering Based On Signal Names

The second type of name-based filter operates on signal names. Here, a filter is specified as a template or pattern. If a signal's hierarchical name contains this pattern, it is excluded from the History file.

Signal-name based filters are activated with the command:

NO HISTORY STRING=*<template>*

The simplest form of ***<template>*** is a character string, possibly quoted. For example, the command:

```
NO HISTORY STRING=and
```

would filter all signals whose names contain the substring "and". Thus, if the circuit contains two instances of the **full-adder** cell illustrated in Figure 2.9-1:

```
p=a  t=full-adder  i=a1,b1,cin1  o=s1,cout1
p=b  t=full-adder  i=a2,b2,cin2  o=s2,cout2
```

then the command:

```
HISTORY LIST:
```

will cause all signals associated with these two macro instances to appear in the History file *except* signals **a.and1**, **a.and2**, **a.and3**, **b.and1**, **b.and2**, and **b.and3**.

A signal name filter can be removed (disabled) by re-specifying it without the **NO** prefix:

HISTORY STRING=*<template>*

Filters can be introduced and removed as necessary. For example, the run command sequence:

```
NO HISTORY STRING=and
HISTORY LIST=a.()
HISTORY STRING=and
HISTORY LIST=b.()
```

causes signals **`a.and1`**, **`a.and2`**, and **`a.and3`** to be absent from the History file, and signals **`b.and1`**, **`b.and2`**, **`b.and3`** to be included.

SIMIC does not assign any significance to the filter template. For example, the command:

```
NO HISTORY STRING=NUMERIC
```

would cause all subsequently specified signals containing the substring "NUMERIC" to be filtered from the signals included in the History file.

Templates such as "and" are not good choices for filters, since this character sequence is likely to be used within part names. For example, this template would cause the signal **`operand.sum`** to be filtered from the History file, since "and" is a substring of "operand". A better method would be to use special characters in the names of internal cell signals. Figure 2.9-2 illustrates a full-adder library cell definition using the "?" character in internal signal names. The appropriate filter would be activated with the command:

```
NO HISTORY STRING="?"
```

```
c= Full-Adder cell
t=full-adder i=a,b,carry-in o=sum,carry-out
   p=xor   t=exor  i=a,b,carry-in    o=sum
   p=and1  t=and   i=a,carry-in      o=?1
   p=and2  t=and   i=b,carry-in      o=?2
   p=and3  t=and   i=a,b             o=?3
   p=or1   t=or    i=and1,and2,and3  o=carry-out
```

Figure 2.9-2 Full-Adder Cell For "?" Signal Name Filtering

The general form of ***<template>*** is:

<repetition> * <character-string>

where ***<repetition>*** is an integer specifying the minimum number of repetitions of the character string ***<character-string>***. Optional spaces may be placed before or after the asterisk. For example:

```
NO HISTORY STRING=2*"?"
```

would cause filtering of subsequently specified signals whose names contain two or more question marks, regardless of their distribution.

As another example:

```
NO HISTORY STRING=2*"."
```

would cause filtering of all signals at a hierarchical depth greater than 2.

Appendix A SIMIC Built-in Primitives

This appendix contains a quick-reference look-up table and description of each SIMIC built-in primitive.

Format of Primitive Descriptions

The description of each primitive contains:

1. a symbol showing its defined pin names and an equivalent logic diagram, where appropriate,
2. a truth table describing its function, where possible,
3. boolean equations describing its function, where possible,
4. a written description, when more information is required than can be conveyed by truth tables or equations,
5. an "equivalent type statement", which would be used in a hypothetical type block to define the primitive's pin names and their order.

In the symbols for the primitives, clock inputs of edge-triggered flip-flops are indicated with a **">"** on the clock input pin. Also a small circle (bubble) indicates an active-low input pin.

When a dash (—) appears in a truth table, it indicates that the corresponding input may have any logic value (0, 1, or X). Unless otherwise stated, input combinations not covered by any truth table row cause an unknown (X) value at each output.

Pin Naming Conventions

Pins of primitives may be grouped into ports. The syntax for fixed-size ports is (angle brackets enclose syntactic items, and are not characters within the port reference):

```
<port_name>[<start>:<end>]
```

where ***<port_name>*** is the name of the port, and ***<start>*** and ***<end>*** are the limits of a consecutive range of pin indices (which may be either ascending or descending). For example:

```
a[1:2]
```

specifies that there are two ordered pins named `a`[1] and `a`[2], and

```
b[2:1]
```

specifies that there are two ordered pins named `b`[2] and `b`[1].

An extended vector notation is used to represent variable-size ports. Its syntax can either be:

```
<port_name>[<from>-<to>:<end>]
```

or

```
<port_name>[<start>:<from>-<to>]
```

where ***<from>-<to>*** represents a range of values for ***<start>*** or ***<end>***, depending on whether the first or second limit is variable. For example:

```
addr[15-0:0]
```

specified that from 1 to 16 signals may be connected to port `addr`. If, for example, four signals are connected, then they will be connected to pins `addr`[3], `addr`[2], `addr`[1], and `addr`[0]. If two signals are connected, then they will be connected to pins `addr`[1] and `addr`[0].

Instantiating Primitives Correctly

Each primitive's "equivalent type statement" defines a template to correctly instantiate it. For example, to instantiate an AANOR element whose inputs are the signals `q`, `r`, `s`, and `t`, and whose output signal, `u`, is the function:

$$\overline{q*r + s*t}$$

the AANOR "equivalent type statement" is referenced:

```
Type=AANOR  i=A[1],A[2],B[1],B[2]  o=Q
```

Using this TYPE statement, a PART statement

```
Part=pname  Type=AANOR  i=q,r,s,t  o=u
```

can be constructed to connect the signals properly by-pin-order.

Alternatively, a PART statement can be constructed to connect the signals properly by-pin-name:

```
Part=pname  Type=AANOR $
   i=q(A[1]),s(B[1]),r(A[2]),t(B[2])  o=u
```

SIMIC BUILT-IN ELEMENTS

Type	Description
SIMPLE COMBINATORIAL GATES	
Type=INV i=I o=Q	Inverter gate
Type=AND i=I[1:1-32767] o=Q	AND gate
Type=NAND i=I[1:1-32767] o=Q	NAND gate
Type=OR i=I[1:1-32767] o=Q	OR gate
Type=NOR i=I[1:1-32767] o=Q	NOR gate
Type=EXOR i=I[1:1-32767] o=Q	Exclusive-OR gate
Type=EXNOR i=I[1:1-32767] o=Q	Exclusive-NOR gate
COMBINATORIAL FUNCTIONS	
Type=AANOR i=A[1:2],B[1:2] o=Q	AND-AND-NOR function
Type=OONAND i=A[1:2],B[1:2] o=Q	OR-OR-NAND function
Type=MUX i=A,B,C o=Q	2-input Multiplexer function
LATCHES	
Type=NANDL i=R,S o=Q	NAND latch
Type=NORL i=R,S o=Q	NOR latch
Type=DL i=NR,NS,C,D o=Q Type=DPL i=NR,NS,C,D o=Q	D latch with positive clock
Type=DNL i=NR,NS,C,D o=Q	D latch with negative clock
FLIP-FLOPS	
Type=DNCF i=NR,NS,C,D o=Q	D flip-flop with negative clock
Type=DCF i=NR,NS,C,D o=Q Type=DPCF i=NR,NS,C,D o=Q	D flip-flop with positive clock
Type=JKCF i=NR,NS,C,J,K o=Q Type=JKNCF i=NR,NS,C,J,K o=Q	JK flip-flop with negative clock
Type=JKPCF i=NR,NS,C,J,K o=Q	JK flip-flop with positive clock
Type=TCF i=NR,NS,T o=Q Type=TNCF i=NR,NS,T o=Q	T flip-flop with negative clock
Type=TPCF i=NR,NS,T o=Q	T flip-flop with positive toggle

SIMIC BUILT-IN ELEMENTS

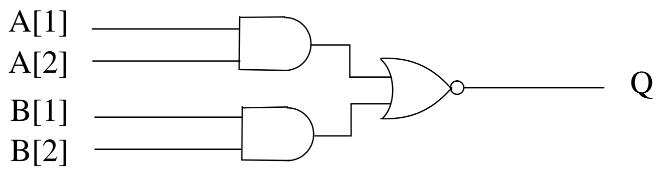
Type	Description
TRISTATE FUNCTIONS	
Type=TGATE i=P,N o=Q	P-N totem tristate driver
Type=TINVN i=EN,D o=Q	Tristate inverter, high enable
Type=TINVP i=EN,D o=Q	Tristate inverter, low enable
Type=TPADN i=EN,D o=Q	Tristate buffer, high enable
Type=TPADP i=EN,D o=Q	Tristate buffer, low enable
PROGRAMMABLE FUNCTIONS AND MEMORIES	
Type=BOOLEAN i=I[1:1-32767] o=Q[1:1-32767] state=STATE[1:1-32767]	User-definable function
Type=ROM i=CS,RE,WE,AE,ADDR[0-15:0] o=DATA[0-99:0]	ROM element
Type=RAMA i=CS,RE,WE,C,ADDR[0-15:0], DATAIN[0-99:0] o=DATA[0-99:0]	RAM element with clocked write
Type=RAMB i=CS,RE,WE,RADDR[0-15:0], WADDR[0-15:0],DATAIN[0-99:0] o=DATA[0-99:0]	RAM (RAMC with separate read and write address lines)
Type=RAMC i=CS,RE,WE,AE,ADDR[0-15:0], DATAIN[0-99:0] o=DATA[0-99:0]	RAM with Address Enable
Type=PLA i=CS,EN,I[1:1-32767] o=Q[1:1-32767]	User-definable PLA element
SWITCHES	
Type=BTGN i=I b=B[1:2]	Ideal Bidirectional switch (alias BTG) enabled when I is logical-1
Type=BTGP i=I b=B[1:2]	Ideal Bidirectional switch enabled when I is logical-0
Type=BTGRN i=I b=B[1:2]	Resistive Bidirectional switch enabled when I is logical-1
Type=BTGRP i=I b=B[1:2]	Resistive Bidirectional switch enabled when I is logical-0
Type=UTGRN i=EN,D o=Q	Resistive Unidirectional switch enabled when I is logical-1
Type=UTGRP i=EN,D o=Q	Resistive Unidirectional switch enabled when I is logical-0

SIMIC BUILT-IN ELEMENTS

Type	Description
BACKANNOTATION ELEMENTS	
Type=DELAY i=I o=Q	Path delay element
Type=NET o=Q	Net loading element

AANOR

Model:



Truth Table:

A[1]	A[2]	B[1]	B[2]	Q
0	—	0	—	1
0	—	—	0	1
—	0	0	—	1
—	0	—	0	1
1	1	—	—	0
—	—	1	1	0

Boolean Description:

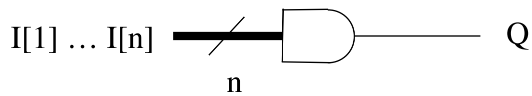
$$Q = \overline{(A[1]*A[2] + B[1]*B[2])}$$
$$= (\overline{A[1]} + \overline{A[2]})*(\overline{B[1]} + \overline{B[2]})$$

Equivalent Type Statement:

Type=AANOR i=A[1],A[2],B[1],B[2] o=Q

AND

Model:



Truth Table:

(For two inputs)

i[1]	i[2]	Q
0	—	0
—	0	0
X	X	X
X	1	X
1	X	X
1	1	1

Boolean Description:

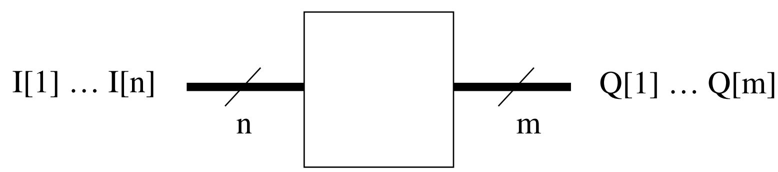
Q = I[1] * I[2] * ... I[n]

Equivalent Type Statement:

Type=AND   i=I[1:1–32767]   o=Q

BOOLEAN

Model:



Truth Table:

(See Text)

Boolean Description:

(See Text)

Equivalent Type Statement:

Type=BOOLEAN i=I[1:1–32767] o=Q[1:1–32767] [state=STATE[1:1–32767]]

Description: BOOLEAN

Boolean equations describe the relationships between the inputs, outputs, and internal state-variables (for sequential elements) of a Boolean element. There should be an equation for every output and state-variable. Typically, these equations are obtained from the network description, but they can be (re)defined afterwards by the SIMIC **CLAMP** command.

All equations associated with a particular Boolean element must be specified within a single **BOOLEAN** keyword-field:

```
BOOLEAN=<boolean_block>
```

where **<boolean_block>** is a **BEGIN/END** block of the form:

```
BEGIN; equation; equation; ... ; END;
```

The continuation character, **$**, is used to continue the equations over multiple lines.

A **boolean part** is a SNL PART statement that references the **BOOLEAN** built-in primitive (**TYPE=BOOLEAN**):

```
p=<part_name> t=boolean i=<input_list> $
    o=<output_list> state=<state_list> $
    boolean=<boolean_block>
```

A **boolean type** is a SNL TYPE statement that includes boolean equations (**BOOLEAN=**) or explicitly declares the statement as a boolean type (**COMPOSITION=BOOLEAN**):

```
t=<type_name> i=<input_list> $
    o=<output_list> state=<state_list> $
    boolean=<boolean_block>
```

All instances of the same boolean type (**<type_name>**) will have the same functionality.

Note: X represents “could be either 0 or 1”, and Z represents “tristating, unknown value”

Each boolean equation within the **<boolean_block>** is a four-valued (0, 1, X, Z) unidirectional assignment of the form:

```
<variable> == <expression>;
```

or

```
<variable> := <expression>;
```

In the first form, the left-hand-side variable will be assigned a value of X if the expression evaluates to Z, whereas in the second form, the Z will be preserved.

Terms or sub-expressions that are common to multiple expressions may be assigned to variables that hold intermediate values. These variables are called *partials*; they may be freely used to eliminate repreated expressions.

A variable on the left-hand-side of an equation must be either an output, a state-variable, or a partial. The right-hand-side expression may contain any combination of inputs, outputs, state-variables, partial variables, and constants (‘0’, ‘1’, ‘X’, ‘Z’—enclosed in either apostrophes or double quotes—representing logical-0, logical-1, unknown, and high-impedance, respectively).

Value Updates

The value of a partial variable term is updated immediately upon computation, and is therefore available for use in the boolean equations that follow.

Outputs and state-variables, however, retain their original values in all expressions, even if a previous equation assigned a new value (concurrent assignment). This can be viewed as a an implicit clocked assignment. The updated values of state-variables will be available the next time the element becomes active (i.e., the next time one of its inputs changes state). The new output values will be available when their respective rise, fall, or decay times elapse.

Initialization

At the beginning of simulation (time = 0, test = 1), all state-variables and outputs are initialized to ‘X’. Partial variables are always initialized to ‘X’ prior to evaluating the boolean expressions.

The Boolean Operators

The supported operators are, in their order of precedence (highest to lowest):

Operators:

Symbol	Operation
^	Unary Complement
=, ^=	EQUAL, NOT-EQUAL Comparison
*	AND
+, @, ^@	OR, Exclusive-OR, Exclusive-NOR

The binary (i.e., defined for two operands) EQUAL (NOT-EQUAL) comparison operators are ‘1’ if the two operands being compared have equal (unequal) values (‘0’, ‘1’, ‘X’, ‘Z’), and ‘0’ otherwise.

The binary Exclusive-OR and Exclusive-NOR operators can be expressed in terms of the other operators:

```
A @ B     is equivalent to     ^A*B + A*^B
A ^@ B    is equivalent to     ^A*^B + A*B
```

Truth Tables for Boolean Element Operators

Unary Complement

^	
0	1
1	0
X	X
Z	X

Equal

=	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

Not-Equal

^=	0	1	X	Z
0	0	1	1	1
1	1	0	1	1
X	1	1	0	1
Z	1	1	1	0

AND

*	0	1	X	Z
0	0	0	0	0
1	0	1	X	Z
X	0	X	X	X
Z	0	Z	X	Z

OR

+	0	1	X	Z
0	0	1	X	Z
1	1	1	1	1
X	X	1	X	X
Z	Z	1	X	Z

Exclusive-Or

@	0	1	X	Z
0	0	1	X	Z
1	1	0	X	X
X	X	X	X	X
Z	Z	X	X	X

Exclusive-Nor

^@	0	1	X	Z
0	1	0	X	X
1	0	1	X	Z
X	X	X	X	X
Z	X	Z	X	X

Operators of equal precedence are evaluated from left to right. Parenthesis may be used to alter the order of evaluation. For example:

```
^a + b @ c * d
```

is evaluated as:

```
((^a) + b) @ (c * d)
```

If this expression is parenthesized as:

```
(^a) + (b @ (c * d))
```

the resulting value would be identical to a right to left evaluation.

Example – Definition of a Boolean Type

The following TYPE statement defines a master-slave positive-edge triggered D flip-flop. The reset (**nr**) and set (**ns**) asynchronous inputs are active-low. On the falling edge of clock (**c**), the state of the **d** input is transferred to the master rank. On the rising edge of clock, the state of the master rank is transferred to the slave rank. (Note: this example is for illustrative purposes only—most D flip-flops do not operate in ths manner.) If the **nr** and **ns** inputs go from the 00 to the 11 state simultaneously, then a timing problem exists and the flip-flop state becomes unknown.

```
TYPE=dffe I=nr,ns,c,d O=q,nq $
STATE=prev_c,prev_r,prev_s,m BOOLEAN= $
BEGIN; $
clk_rise == c*^prev_c; $
clk_fall == ^c*prev_c; $
tm == (d*clk_fall + m*^clk_fall)*nr + ^ns; $
tq == (m*clk_rise + q*^clk_rise)*nr + ^ns; $
rs_race == (prev_r = '0')*(prev_s = '0')* $
           (nr = '1')*(ns = '1'); $
m == tm*^rs_race + 'X'*rs_race; $
q == tq*^rs_race + 'X'*rs_race; $
nq == ^tq*^rs_race + 'X'*rs_race; $
prev_c == c; $
prev_r == nr; $
prev_s == ns; $
END;
```

In this definition, the following variables are state-variables:

1. **prev_c** -- remembers the previous value of **c**.
2. **prev_r** -- remembers the previous value of **nr**.
3. **prev_s** -- remembers the previous value of **ns**.

and the following variables are partial terms:

1. **`clk_fall`** (becomes 1 if **`c`** executes a 1→0 transition).
2. **`clk_rise`** (becomes 1 if **`c`** executes a 0→1 transition).
3. **`rs_race`**(becomes 1 if **`nr`**,**`ns`** went from 0,0 to 1,1 simultaneously).
4. **`tm`** (if no race, the new value for **`m`**, the master rank state variable).
5. **`tq`** (if no race, the new value for **`q`**, the slave rank state variable).

A typical PART statement that instantiates this **`dffe`** is:

```
PART=df1 TYPE=dffe I=re,se,cl,da O=q,notq
```

Note that no state-variables are referenced in the PART statement, which specifies signal/pin connections, since state-variables are *internal* to the **`dffe`**.

Modifying Boolean Types at Run Time

When debugging a design, it is useful to be able to modify a BOOLEAN definition during the simulation session. This is accomplished with the CLAMP command in one of the following formats:

```
CLAMP TYPE=<type_name> BOOLEAN=<boolean_block>
CLAMP PART=<part_name> BOOLEAN=<boolean_block>
```

where:

- ***<type_name>*** is the name of the boolean TYPE to modify.
- ***<boolean_block>*** is the ***BEGIN; equation; equation; ... END;*** format as described above.
- ***<part_name>*** is the name of a BOOLEAN instance to modify.

If *all* instances of a BOOLEAN definition are to be changed, the first form of the **`CLAMP`** command is used. If the BOOLEAN function was defined with a TYPE statement (boolean type), then ***<type_name>*** is the type's name as specified in the **`TYPE`** keyword-field. Alternatively, if the BOOLEAN function was defined with a PART statement (boolean part), then ***<type_name>*** consists of the name of the macro containing the part, followed by a dot (.), followed by the PART's instance name. For example, in the following macro:

```
Type=buf I=a O=b
Part=q I=a O=b BOOLEAN=BEGIN;b == a; END;
```

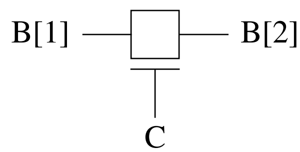
<type_name> would be **`buf.q`**. The **`CLAMP TYPE`** command will replace all instances of this type with the new equations.

If only a *specific* instance of a BOOLEAN type requires modification, then the second form of the **CLAMP** command is used. Here, the **PART** keyword-field specifies the BOOLEAN's instance name.

Note that the number of inputs, outputs, or state-variables cannot be changed with the **CLAMP** command. This requires a change in the network description and a subsequent re-compilation of the circuit.

BTGN and BTGP

Model:



Truth Table:

(See Text)

Boolean Description:

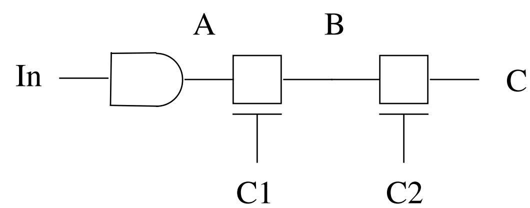
(See Text)

Equivalent Type Statement:

Type=BTGP i=C b=B[1],B[2]
Type=BTGN i=C b=B[1],B[2]

Description:

The BTGN and BTGP primitives are ideal switches having zero resistance and delay. They can be viewed as instantaneous relays, shorting the two ports B[1] and B[2] whenever the control, C, is enabled. The BTGN primitive is positive enabled on the C input (enabled with a logical-1), whereas the BTGP is negative enabled (enabled with a logical-0). Loading on nodes dynamically shorted together is reflected back to the source drivers, and their delays are modified accordingly. For example:



Assuming that nodes **`A`**, **`B`**, and **`C`** each have 1 unit of load, then the driver at the end of the chain would see:

- 1 Unit of load when **`C1`** is disabled, since only node **`A`** is directly connected to the driver to contributes to the loading.
- 2 Units of load if **`C1`** is enabled and **`C2`** is disabled, since both nodes **`A`** and **`B`** contribute to the loading.
- 3 Units of load if both switches are enabled.

Disabling Dynamic Delay Computation

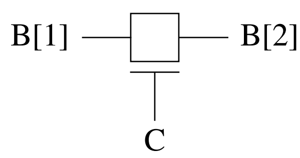
By default, dynamic delay computation is enabled. If you do not want to propagate loading through dynamically connected nodes (and have driver delays modified accordingly), this feature can be disabled by issuing the run command:

```
NO SIMULATE BTGDELAY:
```

This will force all nodes to maintain their isolated loading values, even when connected by enabled BTGNs or BTGPs.

BTGRN and BTGRP

Model:



Truth Table:

(See Text)

Boolean Description:

(See Text)

Equivalent Type Statement:

Type=BTGRP i=C b=B[1],B[2] sdepth=1
Type=BTGRN i=C b=B[1],B[2] sdepth=1

Description:

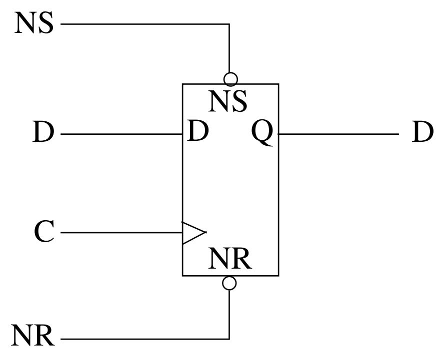
The BTGRN and BTGRP primitives are non-ideal switch elements. ON-resistance is specified by assigning a SERIES-DEPTH (SDEPTH) value upon instantiating the switch in a SNL PART statement. The SDEPTH value can range from 1 to 32766, where 1 is the minimum resistance and 32766 is the maximum resistance of the switch. If unspecified, the SDEPTH value is defaulted to 1. The BTGRN is enabled when its control input, C, is a logical-1, while the BTGRP is enabled when C is a logical-0.

It is important that charge-storage be modeled reasonably in switch network (i.e. decays are larger than delays), otherwise the simulation could produce transient X pulses, slowing down the simulation throughput and possibly generating oscillations during network value convergence.

SIMIC optimizes switch level networks during compilation, merging paralleled gates and converting bidirectional BTGRNs and BTGRPs to their respective unidirectional counterparts, UTGRNs and UTGRPs, where possible.

DCF, DPCF

Model:



Truth Table:

NR	NS	C	M	Q
—	0	—	1	1
0	1	—	0	0
1	1	0	D	Q_0
1	1	0→1	M_0	M_0

Boolean Description:

$$M=(\overline{C}*D + C*M_0 + D*M_0)*NR + \overline{NS}$$

$$Q=(C*M_0 + \overline{C}*Q_0 + M_0*Q_0)*NR + \overline{NS}$$

where M_0 is the previous master-rank state, and Q_0 is the previous slave-rank state.

Note: Q changes on the rising clock edge.

Equivalent Type Statement:

Type=DCF i=NR,NS,C,D o=Q

Type=DPCF i=NR,NS,C,D o=Q

Description:

DCF (**DPCF**) is a positive-edge master-slave D flip-flop. Its set (**NS**) and reset (**NR**) inputs are active-low, and the set dominates when both are low. The **DCF** is modeled as a master-slave flip-flop using two state variables: master = **M** and slave = **Q**. The slave is loaded from the master on the rising edge of clock, **C**, while the **D** input is enabled when clock is low.

During circuit compilation, the master-rank is assigned a delay equal to the unloaded slave delay.

Timing checks

Timing checks can be assigned by a **TIMING–CHECKS** block. The timing checks that the **DCF** supports are:

- **SETUP** – this check specifies the duration that an input must be stable *prior to* an active clock edge(rise). Three setup checks are available: setup from data (**SETUP.D**), setup from trailing edge of reset when data=1 (**SETUP.NR**), and setup from trailing edge of set when data=0 (**SETUP.NS**)
- **HOLD** – this check specifies the duration that an input must be stable *after* an active clock edge (rise). Three hold checks are supported: hold from data (**HOLD.D**), hold from reset (**HOLD.NR**), and hold from set (**HOLD.NS**)
- **PULSE–WIDTHS –** this check specifies the minimum width of a pulse on the set, reset, or clock lines. The pulse-widths supported are: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

Unspecified timing checks default to 0 (disabled).

For example:

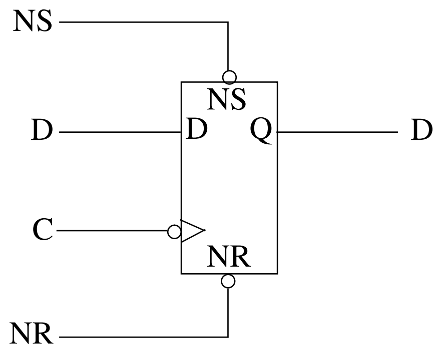
```
part=FF1 type=DCF i=reset,set,clk,data o=q1 $
     timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD.D = 10; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

The above specifies:

1. All setups are 5 units.
1. All holds are 5 units, except hold from **D** which is 10.
1. All pulse-widths are 4, except clock-low pulse-width is 3.

DNCF

Model:



Truth Table:

NR	NS	C	M	Q
—	0	—	1	1
0	1	—	0	0
1	1	1	D	Q_0
1	1	1→0	M_0	M_0

Boolean Description:

$$M=(C*D + \overline{C}*M_0 + D*M_0)*NR + \overline{NS}$$

$$Q=(\overline{C}*M_0 + C*Q_0 + M_0*Q_0)*NR + \overline{NS}$$

where M_0 is the previous master-rank state, and Q_0 is the previous slave-rank state.

Note: Q changes on the falling clock edge.

Equivalent Type Statement:

Type=DNCF i=NR,NS,C,D o=Q

Description:

DNCF is a negative-edge master-slave D flip-flop. Its set (**NS**) and reset (**NR**) inputs are active-low, and the set dominates when both are low. The **DNCF** is modeled as a master-slave flip-flop using two state variables: master = **M** and slave = **Q**. The slave is loaded from the master on the falling edge of clock, **C**, while the **D** input is enabled when clock is high.

During circuit compilation, the master-rank is assigned a delay equal to the unloaded slave delay.

Timing checks

Timing checks can be assigned by a **TIMING–CHECKS** block. The timing checks that the **DNCF** supports are:

- **SETUP** – this check specifies the duration that an input must be stable *prior to* an active clock edge(fall). Three setup checks are available: setup from data (**SETUP.D**), setup from trailing edge of reset when data=1 (**SETUP.NR**), and setup from trailing edge of set when data=0 (**SETUP.NS**)
- **HOLD** – this check specifies the duration that an input must be stable *after* an active clock edge (fall). Three hold checks are supported: hold from data (**HOLD.D**), hold from reset (**HOLD.NR**), and hold from set (**HOLD.NS**)
- **PULSE–WIDTHS** – this check specifies the minimum width of a pulse on the set, reset, or clock lines. The pulse-widths supported are: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

Unspecified timing checks default to 0 (disabled).

For example:

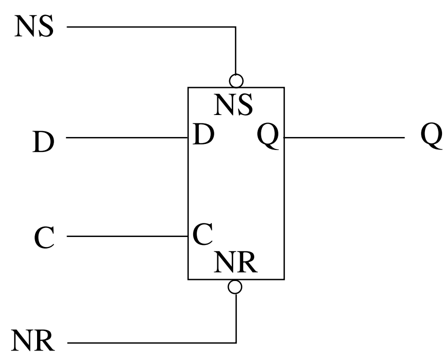
```
part=FF1 type=DNCF i=reset,set,clk,data o=q1 $
     timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD.D = 10; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

The above specifies:

1. All setups are 5 units.
1. All holds are 5 units, except hold from **D** which is 10.
1. All pulse-widths are 4, except clock-low pulse-width is 3.

DL, DPL

Model:



Truth Table:

NR	NS	C	Q
—	0	—	1
0	1	—	0
1	1	1	D
1	1	0	Q_0

Boolean Description:

$$Q=(C*D + \overline{C}*Q_0 + D*Q_0)*NR + \overline{NS}$$

where Q_0 is the previous output state.

Note: D propagates to Q when clock is high, and Q latches when clock is low.

Equivalent Type Statement:

Type=DL i=NR,NS,C,D o=Q Type=DPL i=NR,NS,C,D o=Q

Description:

DL (**DPL**) is a level-sensitive D latch. Its set (**NS**) and reset (**NR**) inputs are active-low, and the set dominates when both are low. When clock, **C**, is high, the **D** input propagates to the **Q** output. When clock is low, the **D** input is disabled, and the latch retains its last-driven **D** value.

Timing checks

Timing checks can be assigned by a **TIMING–CHECKS** block. The timing checks that the **DL** supports are:

- **SETUP** – this check specifies the duration that an input must be stable *prior to* an active clock change (fall). Three setup checks are available: setup from data (**SETUP.D**), setup from trailing edge of reset when data=1 (**SETUP.NR**), and setup from trailing edge of set when data=0 (**SETUP.NS**)
- **HOLD** – this check specifies the duration that an input must be stable *after* an active clock change (fall). Three hold checks are supported: hold from data (**HOLD.D**), hold from reset (**HOLD.NR**), and hold from set (**HOLD.NS**)
- **PULSE–WIDTHS** – this check specifies the minimum width of a pulse on the set, reset, or clock lines. The pulse-widths supported are: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

Unspecified timing checks default to 0 (disabled).

For example:

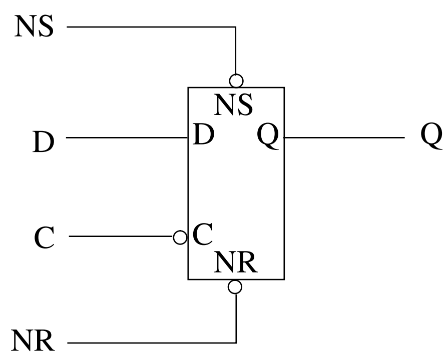
```
part=FF1 type=DL i=reset,set,clk,data o=q1 $
     timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD.D = 10; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

The above specifies:

1. All setups are 5 units.
1. All holds are 5 units, except hold from **D** which is 10.
1. All pulse-widths are 4, except clock-low pulse-width is 3.

DNL

Model:



Truth Table:

NR	NS	C	Q
—	0	—	1
0	1	—	0
1	1	0	D
1	1	1	Q_0

Boolean Description:

$$Q=(\overline{C}*D + C*Q_0 + D*Q_0)*NR + \overline{NS}$$

where Q_0 is the previous output state.

Note: D propagates to Q when clock is low, and Q latches when clock is high.

Equivalent Type Statement:

Type=DNL i=NR,NS,C,D o=Q

Description:

DNL is a level-sensitive D latch. Its set (**NS**) and reset (**NR**) inputs are active-low, and the set dominates when both are low. When clock, **C**, is low, the **D** input propagates to the **Q** output. When clock is high, the **D** input is disabled, and the latch retains its last-driven **D** value.

Timing checks

Timing checks can be assigned by a **TIMING–CHECKS** block. The timing checks that the **DNL** supports are:

- **SETUP** – this check specifies the duration that an input must be stable *prior to* an active clock change (rise). Three setup checks are available: setup from data (**SETUP.D**), setup from trailing edge of reset when data=1 (**SETUP.NR**), and setup from trailing edge of set when data=0 (**SETUP.NS**)
- **HOLD** – this check specifies the duration that an input must be stable *after* an active clock change (rise). Three hold checks are supported: hold from data (**HOLD.D**), hold from reset (**HOLD.NR**), and hold from set (**HOLD.NS**)
- **PULSE–WIDTHS –** this check specifies the minimum width of a pulse on the set, reset, or clock lines. The pulse-widths supported are: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

Unspecified timing checks default to 0 (disabled).

For example:

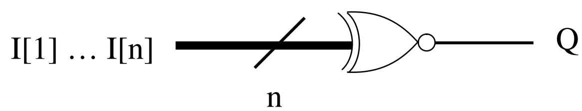
```
part=FF1 type=DNL i=reset,set,clk,data o=q1 $
     timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD.D = 10; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

The above specifies:

1. All setups are 5 units.
1. All holds are 5 units, except hold from **D** which is 10.
1. All pulse-widths are 4, except clock-low pulse-width is 3.

EXNOR

Model:



Truth Table:

(For two inputs)

I[1]	I[2]	Q
0	0	1
0	1	0
1	0	0
1	1	1

Boolean Description:

$$Q = \overline{I[0] \oplus I[2] ... \oplus I[n]}$$

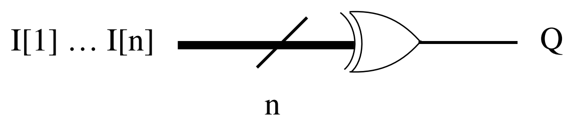
Q is high when an even number of inputs are high, low when an odd number of inputs are high, and X when any input is X

Equivalent Type Statement:

Type=EXNOR i=i[1:1-32767] o=Q

EXOR

Model:



Truth Table:

(For two inputs)

I[1]	I[2]	Q
0	0	0
0	1	1
1	0	1
1	1	0

Boolean Description:

$$Q = I[0] \oplus I[2] ... \oplus I[n]$$

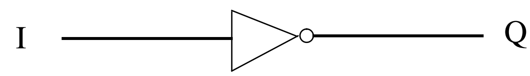
Q is high when an odd number of inputs are high, low when an even number of inputs are high, and X when any input is X

Equivalent Type Statement:

Type=EXOR i=I[1:1-32767] o=Q

INV

Model:



Truth Table:

(For two inputs)

I	Q
0	1
1	0

Boolean Description:

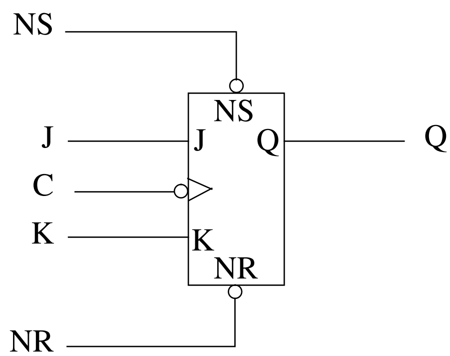
$Q = \overline{I}$

Equivalent Type Statement:

Type=INV i=I o=Q

JKCF, JKNCF

Model:



Truth Table:

NR	NS	J	K	C	M	Q
—	0	—	—	—	1	1
0	1	—	—	—	0	0
1	1	0	0	1	Q_0	Q_0
1	1	0	1	1	0	Q_0
1	1	1	0	1	1	Q_0
1	1	1	1	1	$\overline{Q}_0$	Q_0
1	1	—	—	1→0	M_0	M_0

Boolean Description:

$D=J*\overline{Q}_0 + \overline{K}*Q_0 + J*\overline{K}$

$M=(C*D + \overline{C}*M_0 + M_0*D)*NR + \overline{NS}$

$Q=(\overline{C}*M_0 + C*Q_0 + M_0*Q_0)*NR + \overline{NS}$

where M_0 is the previous master-rank and Q_0 is the previous slave-rank.

Note: Q changes on the falling edge of clock.

Equivalent Type Statement:

Type=JKCF i=NR,NS,C,J,K o=Q

Type=JKNCF i=NR,NS,C,J,K o=Q

Description:

JKCF (**JKNCF**) is a negative-edge master-slave JK flip-flop, using two state variables: master = **M** and slave = **Q**. The slave is loaded from the master on the falling edge of clock. The set (**NS**) and reset (**NR**) inputs are active-low, and the set dominates.

During circuit compilation, the master-rank is assigned a delay equal to the unloaded slave delay.

Timing checks

Timing checks can be assigned by a **TIMING–CHECKS** block. The timing checks that the **JKCF** supports are:

- **SETUP** – this check specifies the duration that an input must be stable *prior* to an active clock edge (fall). Four setup checks are available: setup from J (**SETUP.J**), setup from K (**SETUP.K**), setup from trailing edge of reset when logic-1 is clocked-in (**SETUP.NR**), and setup from trailing edge of set when logic-0 is clocked-in (**SETUP.NS**)
- **HOLD** – this check specifies the duration that an input must be stable *after* an active clock edge fall). Four hold checks are supported: hold from J (**HOLD.J**), hold from K (**HOLD.K**), hold from reset (**HOLD.NR**), and hold from set (**HOLD.NS**)
- **PULSE–WIDTHS** – this check specifies the minimum width of a pulse on the set, reset, or clock lines. The pulse-widths supported are: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

Unspecified timing checks default to 0 (disabled).

For example:

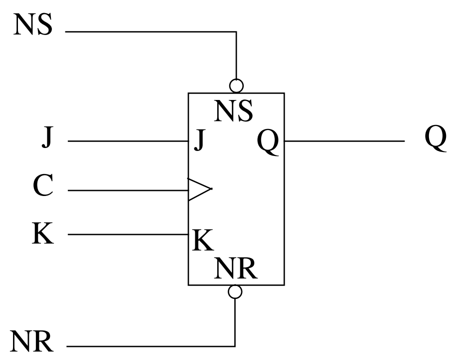
```
part=FF1 type=JKCF i=reset,set,clk,j,k o=q1 $
     timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD.J = 10; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

The above specifies:

1. All setups are 5 units.
1. All holds are 5 units, except hold from j which is 10.
1. All pulse-widths are 4, except clock-low pulse-width is 3.

JKPCF

Model:



Truth Table:

NR	NS	J	K	C	M	Q
—	0	—	—	—	1	1
0	1	—	—	—	0	0
1	1	0	0	0	Q_0	Q_0
1	1	0	1	0	0	Q_0
1	1	1	0	0	1	Q_0
1	1	1	1	0	$\overline{Q}_0$	Q_0
1	1	—	—	0→1	M_0	M_0

Boolean Description:

$D=J*\overline{Q}_0 + \overline{K}*Q_0 + J*\overline{K}$

$M=(\overline{C}*D + C*M_0 + M_0*D)*NR + \overline{NS}$

$Q=(C*M_0 + \overline{C}*Q_0 + M_0*Q_0)*NR + \overline{NS}$

where M_0 is the previous master-rank and Q_0 is the previous slave-rank.

Note: Q changes state on the rising edge of clock.

Equivalent Type Statement:

Type=JKPCF i=NR,NS,C,J,K o=Q

Description:

JKPCF is a positive-edge master-slave JK flip-flop, using two state variables: master = **M** and slave = **Q**. The slave is loaded from the master on the rising edge of clock. The set (**NS**) and reset (**NR**) inputs are active-low, and the set dominates.

During circuit compilation, the master-rank is assigned a delay equal to the unloaded slave delay.

Timing checks

Timing checks can be assigned by a **TIMING–CHECKS** block. The timing checks that the **JKPCF** supports are:

- **SETUP** – this check specifies the duration that an input must be stable *prior* to an active clock edge (rise). Four setup checks are available: setup from J (**SETUP.J**), setup from K (**SETUP.K**), setup from trailing edge of reset when logic-1 is clocked-in (**SETUP.NR**), and setup from trailing edge of set when logic-0 is clocked-in (**SETUP.NS**)
- **HOLD** – this check specifies the duration that an input must be stable *after* an active clock edge (rise). Four hold checks are supported: hold from J (**HOLD.J**), hold from K (**HOLD.K**), hold from reset (**HOLD.NR**), and hold from set (**HOLD.NS**)
- **PULSE–WIDTHS** – this check specifies the minimum width of a pulse on the set, reset, or clock lines. The pulse-widths supported are: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

Unspecified timing checks default to 0 (disabled).

For example:

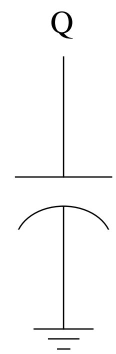
```
part=FF1 type=JKPCF i=reset,set,clk,j,k o=q1 $
     timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD.J = 10; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

The above specifies:

1. All setups are 5 units.
1. All holds are 5 units, except hold from j which is 10.
1. All pulse-widths are 4, except clock-low pulse-width is 3.

LOAD

Model:



Truth Table:

(Not Applicable)

Boolean Description:

The LOAD element is used for backannotation; it adds loading to the specified net.

For example:
`p=sig5 t=load olod=7`

and
`p=cap t=load o=sig5 olod=7`

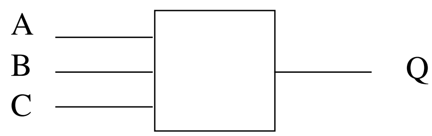
both add 7 units of loading to net **`sig5`**. If the output keyword-field is present, as shown in the second statement above, the name used in the part keyword-field is arbitrary (unlike all other instantiations).

Equivalent Type Statement:

Type= LOAD o=Q olod=0

MUX

Model:



Truth Table:

A	B	C	Q
—	0	0	0
—	1	0	1
0	—	1	0
1	—	1	1
0	0	—	0
1	1	—	1

Boolean Description:

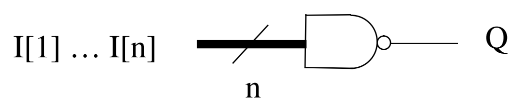
$$Q=A*C + B*\overline{C} + A*B$$

Equivalent Type Statement:

Type=MUX i=A,B,C o=Q

NAND

Model:



Truth Table:

(For two inputs)

I[1]	I[2]	Q
0	—	1
—	0	1
X	X	X
X	1	X
1	X	X
1	1	0

Boolean Description:

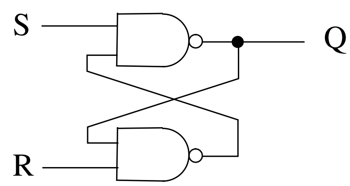
$$Q = \overline{I[1] * I[2] * ... I[n]}$$

Equivalent Type Statement:

Type=NAND i=I[1:1-32767] o=Q

NANDL

Model:



Truth Table:

R	S	Q
—	0	1
0	1	0
1	1	Q_0

Boolean Description:

$$Q = \overline{S} + R*Q_0$$

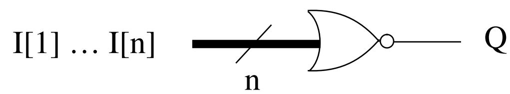
where Q_0 is the previous output state.

Equivalent Type Statement:

Type=NANDL i=R,S o=Q

NOR

Model:



Truth Table:

(For two inputs)

i[1]	i[2]	Q
1	—	0
—	1	0
X	X	X
X	0	X
0	X	X
0	0	1

Boolean Description:

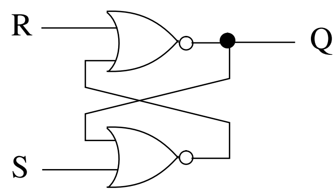
$$Q = \overline{I[1] + I[2] + ...\ I[n]}$$

Equivalent Type Statement:

Type=NOR i=I[1:1-32767] o=Q

NORL

Model:



Truth Table:

R	S	Q
1	—	0
0	1	1
0	0	Q_0

Boolean Description:

$$Q = \overline{R} * (S + Q_0)$$

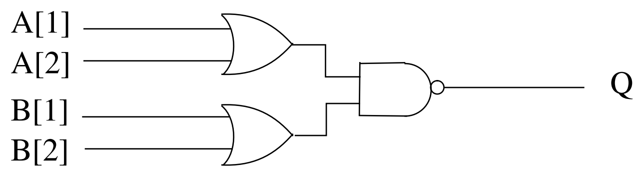
where Q_0 is the previous output state.

Equivalent Type Statement:

Type=NORL i=R,S o=Q

OONAND

Model:



Truth Table:

A[1]	A[2]	B[1]	B[2]	Q
1	—	1	—	0
1	—	—	1	0
—	1	1	—	0
—	1	—	1	0
0	0	—	—	1
—	—	0	0	1

Boolean Description:

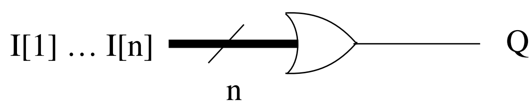
$$
\begin{aligned}
Q &= \overline{(A[1] + A[2])*(B[1] + B[2])} \\
&= \overline{A[1]}*\overline{A[2]} + \overline{B[1]}*\overline{B[2]}
\end{aligned}
$$

Equivalent Type Statement:

Type=OONAND i=A[1],A[2],B[1],B[2] o=Q

OR

Model:



Truth Table:

(For two inputs)

i[1]	i[2]	Q
1	—	1
—	1	1
X	X	X
X	0	X
0	X	X
0	0	0

Boolean Description:

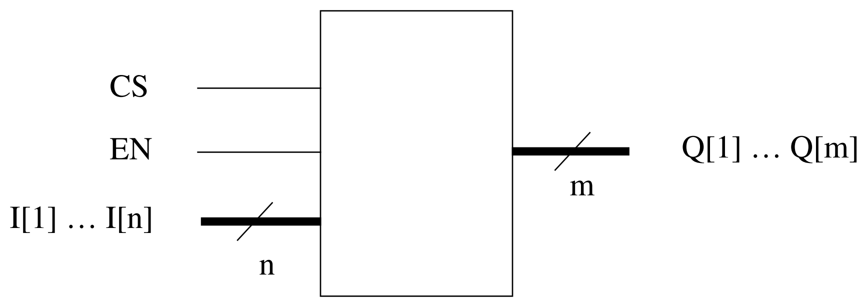
Q = I[1] + I[2] + ... I[n]

Equivalent Type Statement:

Type=OR i=I[1:1–32767] o=Q

PLA

Model:



Truth Table:

CS	EN	Q
0	—	Z
1	0	<disabled>
X	—	X
—	X	X
1	1	<function>

Boolean Description:

(See Text)

Equivalent Type Statement:

Type=PLA i=CS,EN,I[1:1-32767] o=Q[1:1-32767]

Description:

The PLA primitive models an AND/OR plane implementation of a PLA. The PLA's data inputs are used to generate the AND plane outputs, and the AND plane outputs are used to generate the OR plane outputs, which are also the outputs of the PLA.

The first input, CS, controls whether the output is enabled (driving) or not (tristating). The second input, EN, controls whether to perform the AND/OR function or output a user-specified ***disabled*** value. The remaining PLA inputs are the data inputs.

The personality of the PLA is set by CLAMP run commands.

Setting the Disabled Value

When the EN input is logical-0, the outputs are set to the ***disabled*** value specified by the ENABLE option of the CLAMP command. This value can be either **1** or **0**. If unspecified, the value defaults to **0**. For example:

```
CLAMP PART=pla1 ENABLE=1
```

specifies that all outputs should be logical-1 whenever the enable, EN, is logical-0.

The Connection Plane Maps

The AND and OR planes are personalized by the use of connection maps that specify either a TRUE connection, a COMPLEMENT connection, or NO connection to each specific input line of the plane. SIMIC uses the character **1** for TRUE, **0** for COMPLEMENT, and **X** for NO connection.

Setting the AND Plane

The AND plane personality is set by the CLAMP command as follows:

```
CLAMP PART=<name> AND=<ninputs>*<nproducts> $
BITMAP=<connection_map>
```

where ***<name>*** is the PLA's part name, ***<ninputs>*** is the number of inputs to the AND plane (which must be the number of data inputs), and ***<nproducts>*** is the number of AND plane product terms (and the number of inputs to the OR plane). This keyword-field is used for error checking.

The ***<connection_map>*** contains ***<nproducts>*** items, each containing ***<ninputs>*** **0**, **1**, or **X** characters that define a single product term, ordered according to the PLA data inputs.

Setting the OR Plane

The OR plane personality is set by the CLAMP command as follows:

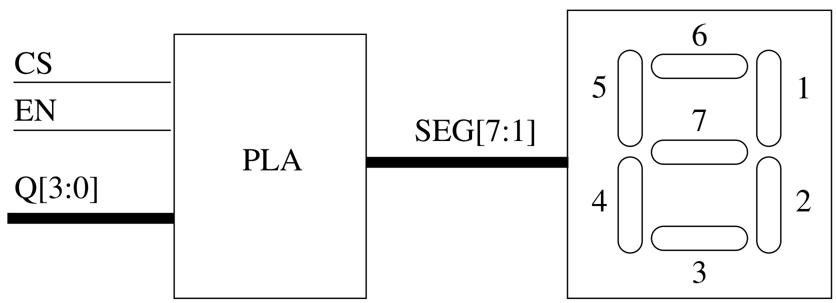
```
CLAMP PART=<name> OR=<nproducts>*<noutputs> $
BITMAP=<connection_map>
```

where ***<name>*** is the PLA's part name, ***<nproducts>*** is the number of inputs to the OR plane (which must be the number of AND plane outputs), and ***<noutputs>*** is the number of OR plane sum terms (and the number of PLA outputs). This keyword-field is used for error checking.

The ***<connection_map>*** contains ***<noutputs>*** items, each containing ***<nproducts>*** **0**, **1**, or **X** entries that define a single sum term, ordered by the AND plane outputs. The sum terms, in turn, are ordered by the PLA outputs.

PLA Example:

The following example illustrates a PLA implementation of a 4-bit to 7-bit decoder to interface a BCD input to a seven-segment decimal digit display:



digit	**Q[3:0]**	**Segment**						
		7	**6**	**5**	**4**	**3**	**2**	**1**
0	0000		ON	ON	ON	ON	ON	ON
1	0001						ON	ON
2	0010	ON	ON		ON	ON		ON
3	0011	ON	ON			ON	ON	ON
4	0100	ON		ON			ON	ON
5	0101	ON	ON	ON		ON	ON	
6	0110	ON	ON	ON	ON	ON	ON	
7	0111		ON				ON	ON
8	1000	ON	ON	ON	ON	ON	ON	ON
9	1001	ON	ON	ON			ON	ON

Assuming that the *i*-th segment ($1 \le i \le 7$) is ON when SEG[*i*] is logical ONE, and is OFF when SEG[*i*] is logical ZERO, and that the remaining six states of Q[3:0] can be treated as *don't-cares*, then the following equations are one realization of the decoder:

$$SEG[7] = Q[3] + Q[2]*\overline{Q[1]} + Q[2]*\overline{Q[0]} + \overline{Q[2]}*Q[1]$$

$$SEG[6] = Q[3] + Q[1] + \overline{Q[2]}*\overline{Q[0]} + Q[2]*\overline{Q[1]}*Q[0]$$

$$SEG[5] = Q[3] + \overline{Q[1]}*\overline{Q[0]} + Q[2]*\overline{Q[1]} + Q[2]*\overline{Q[0]}$$

$$SEG[4] = \overline{Q[2]}*\overline{Q[0]} + Q[1]*\overline{Q[0]}$$

$$SEG[3] = \overline{Q[2]}*\overline{Q[0]} + Q[1]*\overline{Q[0]} + \overline{Q[2]}*Q[1] + Q[2]*\overline{Q[1]}*Q[0]$$

$$SEG[2] = Q[2] + \overline{Q[1]} + Q[0]$$

$$SEG[1] = Q[3] + \overline{Q[1]}*\overline{Q[0]} + Q[1]*Q[0] + \overline{Q[2]}$$

The following table summarizes product term usage in these equations:

Product Term		**Segment**						
		7	**6**	**5**	**4**	**3**	**2**	**1**
1	Q[3]	✓	✓	✓				✓
2	Q[2]						✓	✓*
3	$Q[2]*\overline{Q[1]}$	✓		✓				
4	$Q[2]*\overline{Q[1]}*Q[0]$		✓			✓		
5	$Q[2]*\overline{Q[0]}$	✓		✓				
6	$\overline{Q[2]}*Q[1]$	✓				✓		
7	$\overline{Q[2]}*\overline{Q[0]}$		✓		✓	✓		
8	Q[1]		✓				✓*	
9	Q[1]*Q[0]							✓
10	$Q[1]*\overline{Q[0]}$				✓	✓		
11	$\overline{Q[1]}*\overline{Q[0]}$			✓				✓
12	Q[0]						✓	

Note that there are two entries in the table marked with an asterisk—segment-2's Q[1] term and segment-1's Q[2] term. The segment equations actually require the complements of these terms, which in this example will be realized by entering **0**s in the OR plane. Alternatively, $\overline{Q[1]}$ and $\overline{Q[2]}$ could be generated as two more output terms of the AND plane.

The PLA might be instantiated in the SNL description as:

```
part=pla1 type=pla i=one,one,q[3:0] o=seg[7:1]
```

where the CS and EN inputs have been tied to logic ONE to permanently enable the PLA outputs (of course, other connections to these inputs would be made if this were not desirable).

Personalization of the PLA is accomplished by the two CLAMP commands defining the AND and OR planes; typically, these commands would physically be in a separate file that would be read at run time by an EXECUTE command.

Using the above table, and assuming the PLA's part name is `pla1`, the CLAMP command defining the AND plane would be:

```
clamp part=pla1 and=4*12 bitmap= $
     1xxx x1xx x10x x101 x1x0 x01x x0x0 xx1x $
     xx11 xx10 xx00 xxx1
```

There are 12 product terms, each describing four connections ordered according to the PLA data inputs. For example, the first term, `1xxx`, represents a TRUE connection to Q[3] and NO connection to the other inputs, and therefore is the product term Q[3]. Similarly, the third term, `x10x`, represents NO connection to Q[3], a TRUE connection to Q[2], a COMPLEMENTED connection to Q[1], and NO connection to Q[0], and therefore is the product term Q[2]*$\overline{\text{Q[1]}}$. (Note that the 12 product terms are ordered according to the rows of the usage table above.)

The CLAMP command defining the OR plane would be:

```
clamp part=pla1 or=12*7 bitmap= $
     1x1x11xxxxxx 1xx1xx11xxxx 1x1x1xxxxx1x $
     xxxxxx1xx1xx xxx1x11xx1xx x1xxxxx0xxx1 $
     10xxxxxx1x1x
```

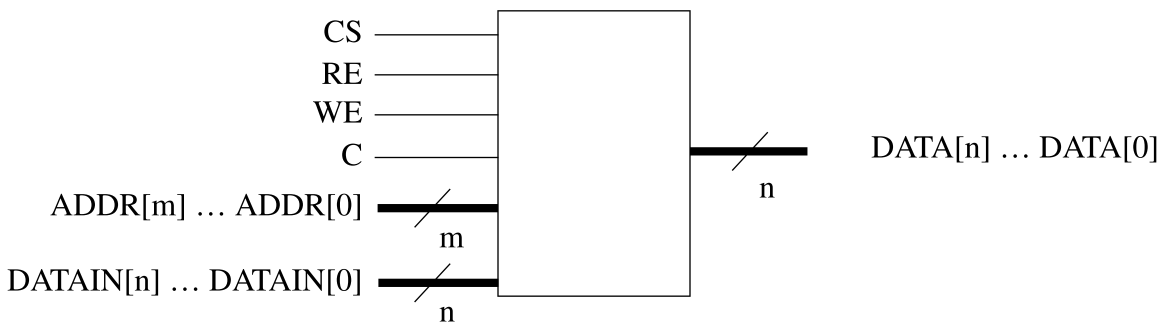
There are 7 sum terms that generate the PLA outputs. The first term corresponds to seg[7], since this is the first signal in the output keyword-field of the above PART statement instantiating the PLA. The other sum terms are ordered accordingly.

Ordering of the 12 connection entries within each sum term follows the AND plane outputs. For example, the first sum term specifies the OR of the TRUE values of the first, third, fifth, and sixth product terms, and NO connection to any of the others. This is exactly the product term usage of segment-7, as indicated in the above table. Similarly, the sixth sum term specifies the OR of the TRUE value of the second product term, the COMPLEMENT of the eighth product term (i.e., $\overline{\text{Q[1]}}$), and the TRUE value of the twelfth produce term. As noted above, this use of the COMPLEMENT in the OR plane eliminated the need to generate $\overline{\text{Q[1]}}$ in the AND plane, since Q[1] was available.

End of Example

RAMA

Model:



Truth Table:

CS	RE	WE	CLK	DATA
0	—	—	—	Z
—	0	—	—	Z
1	X	—	—	X
X	1	—	—	X
X	X	—	—	X
1	1	—	—[1]	contents at locacation ADDR

Boolean Description:

(See Text)

Equivalent Type Statement:

Type=RAMA i=CS,RE,WE,C,ADDR[15-0:0],DATAIN[99-0:0] o=DATA[99-0:0]

1. Note: If the clock rises, and the chip is selected and write enabled, then the contents of the DATAIN bus are written to the current address, specified by ADDR. If the RAMA is simultaneously read enabled, the new contents will be at the DATA outputs (write-before-read).

Description:

This is a single port RAM primitive. The number of DATAIN lines must be equal to the number of DATA lines (therefore, the semicolon between the ADDR and DATAIN inputs is optional in the instantiating PART statement's input field, since the number of DATAIN lines is constrained). Data is written to the RAM on a rising clock edge and CS and WE are high.

Initializing the Contents of a RAM

Normally, RAM contents are initialized during simulation by performing writes to the RAM. However, SIMIC also supports RAM initialization prior to simulation via the CLAMP command. The command structure is:

```
CLAMP PART=<name> DATA=<addresses and data>
```

where ***<name>*** is the RAM's part name, and ***<addresses and data>*** is the data to write into the RAM in the following format:

```
X<address> <data> X<address> <data> ...
```

where ***<address>*** is the starting address for the data that follows, in hexadecimal, and ***<data>*** is the data described in hexadecimal. For example:

```
CLAMP PART=RAM1 DATA= $
X0000     00 01 02 03 04 05 06 07 $
          08 09 0A 0B 0C 0D 0E 0F $
XC000     FF FF FF FF
```

Specifying the Last Valid Address

The last valid address for the RAM can be specified in the SNL description. If a write or read operation occurs beyond this address, then an appropriate error message will be issued. This is done via the LASTADDR keyword in the PART statement. For example if the last valid address is 3000, you would include `LASTADDR=3000` in the PART statement. If unspecified, the last valid address is defaulted to 2^n-1, where n is the number of address lines.

X Address Line Handling

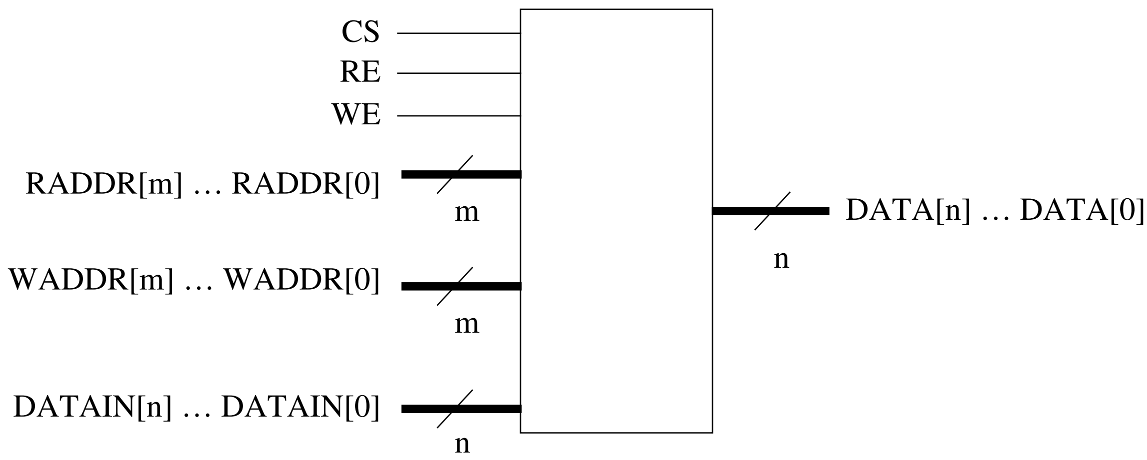
If there are unknowns (X) on address lines during a read or write, then SIMIC will compare the data at each location that could be accessed and set to X only those bits that conflict. By default, if more than 4 address lines are unknown, then SIMIC will abandon this scheme and instead output an X for each output if a read is enabled, and set all RAM locations to X if a write is enabled. This threshold can be set globally by the DEFINE command:

```
DEFINE XADDRESS=<n>
```

where <n> is the maximum number of X address lines to enumerate.

RAMB

Model:



Truth Table:

CS	RE	WE	DATA
0	—	—	Z
—	0	—	Z
1	X	—	X
X	1	—	X
X	X	—	X
1	1	0	contents of RADDR
1	1	1	contents of RADDR[1]

Boolean Description:

(See Text)

Equivalent Type Statement:

Type=RAMB i=CS,RE,WE,RADDR[15-0:0],WADDR[15-0:0],DATAIN[99-0:0] $
o=DATA[99-0:0]

1. Note: If the write address, specified by WADDR, and the read address, specified by RADDR, are identical, then the newly-written contents of the selected address appears at the DATA output (write-before-read).

Description:

This is a dual port RAM primitive. The number of DATAIN lines must be equal to the number of DATA lines and the number of RADDR lines must equal the number of WADDR lines (therefore, the semicolons between the RADDR, WADDR, and DATAIN inputs are optional in the instantiating PART statement's input field, since their dimensions are constrained).

Initializing the Contents of a RAM

Normally, RAM contents are initialized during simulation by performing writes to the RAM. However, SIMIC also supports RAM initialization prior to simulation via the CLAMP command. The command structure is:

```
CLAMP PART=<name> DATA=<addresses and data>
```

where ***<name>*** is the RAM's part name, and ***<addresses and data>*** is the data to write into the RAM in the following format:

```
X<address> <data> X<address> <data> ...
```

where ***<address>*** is the starting address for the data that follows, in hexadecimal, and ***<data>*** is the data described in hexadecimal. For example:

```
CLAMP PART=RAM1 DATA= $
X0000     00 01 02 03 04 05 06 07 $
          08 09 0A 0B 0C 0D 0E 0F $
XC000     FF FF FF FF
```

Specifying the Last Valid Address

The last valid address for the RAM can be specified in the SNL description. If a write or read operation occurs beyond this address, then an appropriate error message will be issued. This is done via the LASTADDR keyword in the PART statement. For example if the last valid address is 3000, you would include `LASTADDR=3000` in the PART statement. If unspecified, the last valid address is defaulted to 2^n-1, where n is the number of address lines.

X Address Line Handling

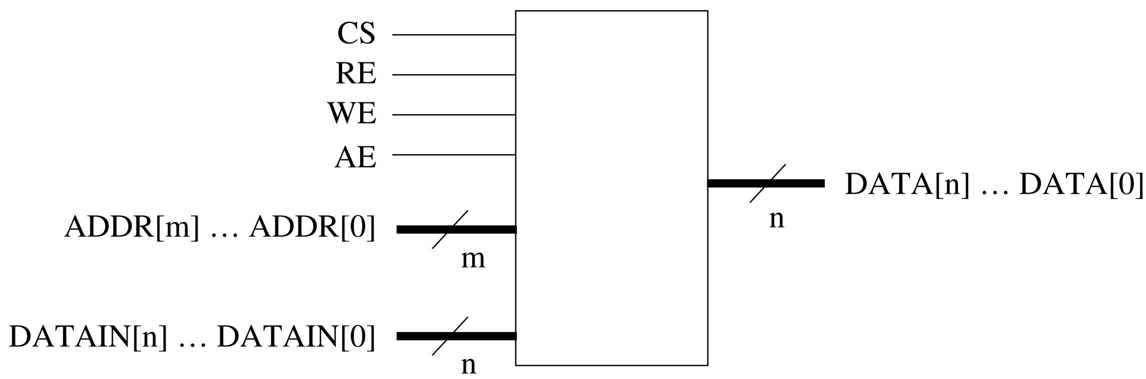
If there are unknowns (X) on address lines during a read or write, then SIMIC will compare the data at each location that could be accessed and set to X only those bits that conflict. By default, if more than 4 address lines are unknown, then SIMIC will abandon this scheme and instead output an X for each output if a read is enabled, and set all RAM locations to X if a write is enabled. This threshold can be set globally by the DEFINE command:

```
DEFINE XADDRESS=<n>
```

where <n> is the maximum number of X address lines to enumerate.

RAMC

Model:



Truth Table:

CS	AE	RE	WE	DATA
0	—	—	—	Z
—	0	—	—	Z
—	—	0	—	Z
1,X	1,X	X	—	X
X	1,X	1,X	—	X
1,X	X	1,X	—	X
1	1	1	0	contents of ADDR
1	1	1	1	new contents of ADDR[1]

Boolean Description:

(See Text)

Equivalent Type Statement:

Type=RAMC i=CS,RE,WE,AE,ADDR[15-0:0],DATAIN[99-0:0] o=DATA[99-0:0]

1.Note: If the RAMC is simultaneously read enabled and write enabled, then the newly-written contents of the selected address appears at the DATA output (write-before-read).

Description:

This is a single port RAM primitive. The number of DATAIN lines must be equal to the number of DATA lines (therefore, the semicolon between the ADDR and DATAIN inputs is optional in the instantiating PART statement's input field, since the number of DATAIN lines is constrained).

Initializing the Contents of a RAM

Normally, RAM contents are initialized during simulation by performing writes to the RAM. However, SIMIC also supports RAM initialization prior to simulation via the CLAMP command. The command structure is:

```
CLAMP PART=<name> DATA=<addresses and data>
```

where ***<name>*** is the RAM's part name, and ***<addresses and data>*** is the data to write into the RAM in the following format:

```
X<address> <data> X<address> <data> ...
```

where ***<address>*** is the starting address for the data that follows, in hexadecimal, and ***<data>*** is the data described in hexadecimal. For example:

```
CLAMP PART=RAM1 DATA= $
X0000     00 01 02 03 04 05 06 07 $
          08 09 0A 0B 0C 0D 0E 0F $
XC000     FF FF FF FF
```

Specifying the Last Valid Address

The last valid address for the RAM can be specified in the SNL description. If a write or read operation occurs beyond this address, then an appropriate error message will be issued. This is done via the LASTADDR keyword in the PART statement. For example if the last valid address is 3000, you would include `LASTADDR=3000` in the PART statement. If unspecified, the last valid address is defaulted to 2^n-1, where n is the number of address lines.

X Address Line Handling

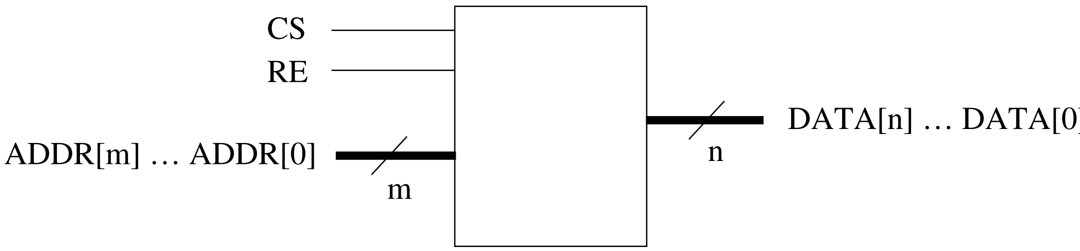
If there are unknowns (X) on address lines during a read or write, then SIMIC will compare the data at each location that could be accessed and set to X only those bits that conflict. By default, if more than 4 address lines are unknown, then SIMIC will abandon this scheme and instead output an X for each output if a read is enabled, and set all RAM locations to X if a write is enabled. This threshold can be set globally by the DEFINE command:

```
DEFINE XADDRESS=<n>
```

where <n> is the maximum number of X address lines to enumerate.

ROM

Model:



Truth Table:

CS	RE	DATA
0	—	Z
—	0	Z
1	X	X
X	1	X
X	X	X
1	1	contents at location ADDR

Boolean Description:

(See Text)

Equivalent Type Statement:

Type=ROM i=CS,RE,ADDR[15-0:0] o=DATA[99-0:0]

Description:

This is a single port ROM primitive.

Initializing the Contents of a ROM

A ROM is initialized prior to simulation via the CLAMP command. The command structure is:

```
CLAMP PART=<name> DATA=<addresses and data>
```

where ***<name>*** is the ROM's part name, and ***<addresses and data>*** is the data to write into the ROM in the following format:

```
X<address> <data> X<address> <data> ...
```

where ***<address>*** is the starting address for the data that follows, in hexadecimal, and ***<data>*** is the data described in hexadecimal. For example:

```
CLAMP PART=ROM1 DATA= $
X0000     00 01 02 03 04 05 06 07 $
          08 09 0A 0B 0C 0D 0E 0F $
XFF00     FF FF FF FF
```

Specifying the Last Valid Address

The last valid address for the ROM can be specified in the SNL description. If a read operation occurs beyond this address, then an appropriate error message will be issued. This is done via the LASTADDR keyword in the PART statement. For example if the last valid address is 3000, you would include `LASTADDR=3000` in the PART statement. If unspecified, the last valid address is defaulted to 2^n-1, where *n* is the number of address lines.

X Address Line Handling

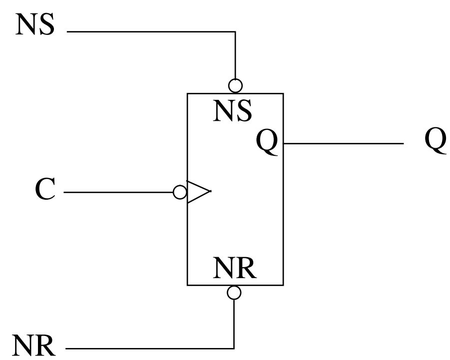
If there are unknowns (X) on address lines during a read, then SIMIC will compare the data at each location that could be accessed and set to X only those bits that conflict. By default, if more than 4 address lines are unknown, then SIMIC will abandon this scheme and instead output an X for each output. This threshold can be set globally by the DEFINE command:

```
DEFINE XADDRESS=<n>
```

where <n> is the maximum number of X address lines to enumerate.

TCF, TNCF

Model:



Truth Table:

NR	NS	C	M	Q
—	0	—	1	1
0	1	—	0	0
1	1	1	$\overline{Q}_0$	Q_0
1	1	1→0	M_0	M_0

Boolean Description:

$$M=(\overline{Q}_0*C + M_0*\overline{C} + M_0*\overline{Q}_0)*NR + \overline{NS}$$

$$Q=(\overline{C}*M_0 + C*Q_0 + M_0*Q_0)*NR + \overline{NS}$$

where M_0 is the previous master-rank state, and Q_0 is the previous slave-rank state.

Note: Q changes on the falling edge of clock.

Equivalent Type Statement:

Type=TCF i=NR,NS,C o=Q

Type=TNCF i=NR,NS,C o=Q

Description:

TCF (**TNCF**) is a negative-edge-triggered master-slave T flip-flop, using two state variables: master = **M** and slave = **Q**. The complement of the slave is loaded into the master when clock is ONE. The master is then loaded into the slave on the falling clock edge. The set (**NS**) and reset (**NR**) inputs are active-low, and the set dominates.

During circuit compilation, the master-rank is assigned a delay equal to the unloaded slave delay.

Timing checks

Timing checks can be assigned by a **TIMING–CHECKS** block. The timing checks that the **TCF** supports are:

- **SETUP** – this check specifies the duration that an input must be stable *prior to* an active clock edge(fall). Two setup checks are available: setup from reset (**SETUP.NR**), and setup from set (**SETUP.NS**)
- **HOLD** – this check specifies the duration that an input must be stable *after* an active clock edge (fall). Two hold checks are supported: hold from reset (**HOLD.NR**), and hold from set (**HOLD.NS**)
- **PULSE–WIDTHS –** this check specifies the minimum width of a pulse on the set, reset, or clock lines. The pulse-widths supported are: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

Unspecified timing checks default to 0 (disabled).

For example:

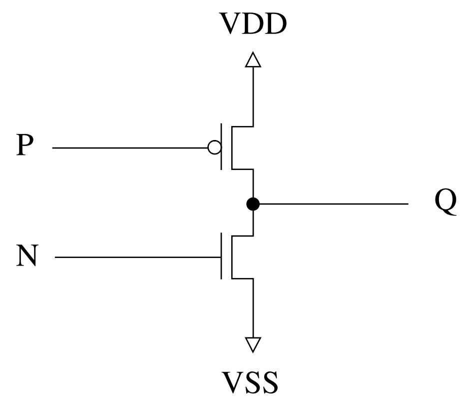
```
part=FF1 type=TCF i=reset,set,clk o=q1 $
     timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

The above specifies:

1. All setups are 5 units.
1. All holds are 5 units.
1. All pulse-widths are 4, except clock-low pulse-width is 3.

TGATE

Model:



Truth Table:

P	N	Q
0	0	1
0	1	X
1	0	Z
1	1	0

Boolean Description:

Equivalent Type Statement:

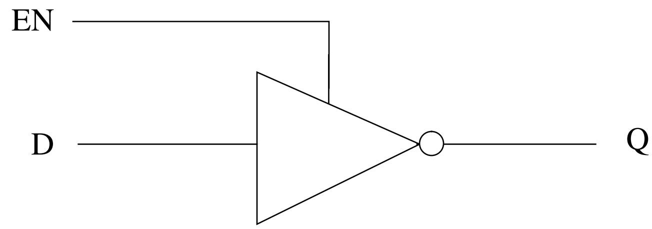
Type=TGATE i=P,N o=Q

Description:

The TGATE primitive models the common totem pole output driver. If both “transistors” are enabled and the high and low drive strengths are equal, and conflict messages are enabled, then a conflict warning is generated.

TINVN

Model:



Truth Table:

EN	Q
0	Z
1	$\overline{D}$

Boolean Description:

Equivalent Type Statement:

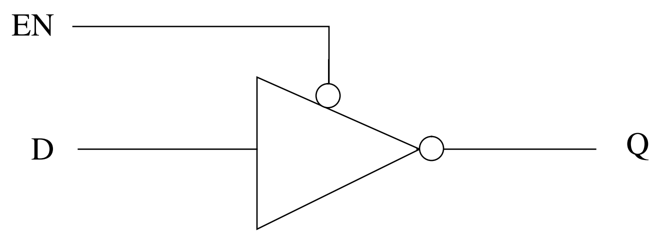
Type=TINVN i=EN,D o=Q

Description:

The TINVN primitive is a tristating inverter with high enable.

TINVP

Model:



Truth Table:

EN	Q
0	$\overline{D}$
1	Z

Boolean Description:

Equivalent Type Statement:

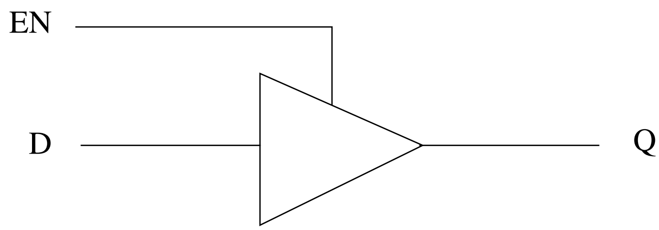
Type=TINVP i=EN,D o=Q

Description:

The TINVP primitive is a tristating inverter with low enable.

TPADN

Model:



Truth Table:

EN	Q
1	D
0	Z

Boolean Description:

Equivalent Type Statement:

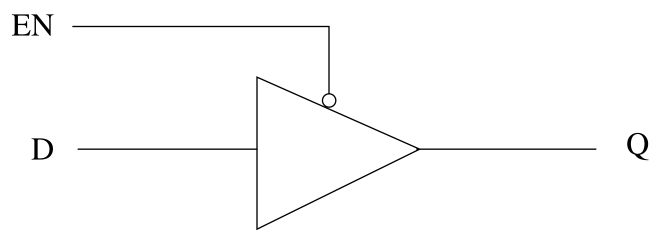
Type=TPADN i=EN,D o=Q

Description:

The TPADN primitive is a non-inverting tristating buffer with high enable.

TPADP

Model:



Truth Table:

EN	Q
0	D
1	Z

Boolean Description:

Equivalent Type Statement:

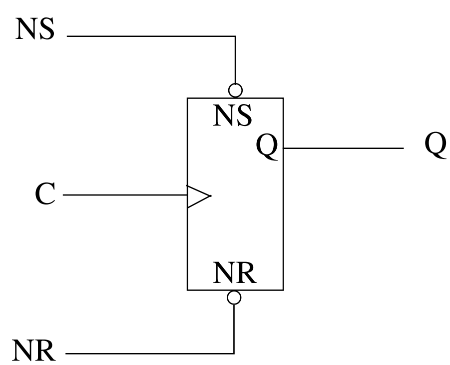
Type=TPADP i=EN,D o=Q

Description:

The TPADP primitive is a non-inverting tristating buffer with low enable.

TPCF

Model:



Truth Table:

NR	NS	C	M	Q
—	0	—	1	1
0	1	—	0	0
1	1	0	$\overline{Q}_0$	Q_0
1	1	$0 \rightarrow 1$	M_0	M_0

Boolean Description:

$$M=(\overline{Q}_0*\overline{C} + M_0*C + M_0*\overline{Q}_0)*NR + \overline{NS}$$

$$Q=(C*M_0 + \overline{C}*Q_0 + M_0*Q_0)*NR + \overline{NS}$$

where M_0 is the previous master-rank state, and Q_0 is the previous slave-rank state.

Note: Q changes on the rising edge of clock.

Equivalent Type Statement:

Type=TPCF i=NR,NS,C o=Q

Description:

TPCF is a positive-edge-triggered master-slave T flip-flop, using two state variables: master = **M** and slave = **Q**. The complement of the slave is loaded into the master when clock is ZERO. The master is then loaded into the slave on the rising clock edge. The set (**NS**) and reset (**NR**) inputs are active-low, and the set dominates.

During circuit compilation, the master-rank is assigned a delay equal to the unloaded slave delay.

Timing checks

Timing checks can be assigned by a **TIMING–CHECKS** block. The timing checks that the **TPCF** supports are:

- **SETUP** – this check specifies the duration that an input must be stable *prior to* an active clock edge(rise). Two setup checks are available: setup from reset (**SETUP.NR**), and setup from set (**SETUP.NS**)
- **HOLD** – this check specifies the duration that an input must be stable *after* an active clock edge (rise). Two hold checks are supported: hold from reset (**HOLD.NR**), and hold from set (**HOLD.NS**)
- **PULSE–WIDTHS –** this check specifies the minimum width of a pulse on the set, reset, or clock lines. The pulse-widths supported are: pulse-width reset (**PW.NR**), pulse-width set (**PW.NS**), and high and low pulse-width clock (**PW.C.H** and **PW.C.L** respectively).

Unspecified timing checks default to 0 (disabled).

For example:

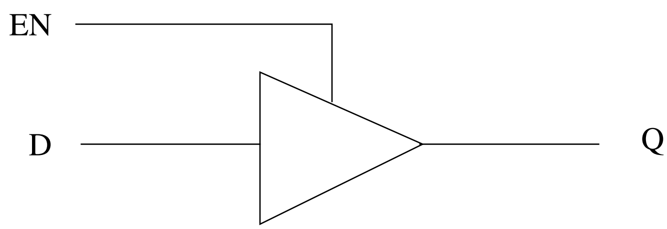
```
part=FF1 type=TPCF i=reset,set,clk o=q1 $
     timing–checks= $
          BEGIN; $
          SETUP = 5; $
          HOLD = 5; $
          PW = 4; $
          PW.C.L = 3; $
          END;
```

The above specifies:

1. All setups are 5 units.
1. All holds are 5 units.
1. All pulse-widths are 4, except clock-low pulse-width is 3.

UTGRN

Model:



Truth Table:

EN	Q
1	D
0	Z

Boolean Description:

Equivalent Type Statement:

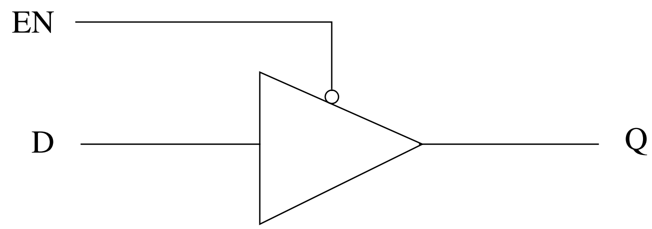
Type=UTGRN i=EN,D o=Q

Description:

The UTGRN primitive is identical to the BTGRN primitive, but only provides unidirectional flow. The circuit compiler automatically converts BTGRN elements to UTGRN elements where possible, to improve simulation throughput.

UTGRP

Model:



Truth Table:

EN	Q
0	D
1	Z

Boolean Description:

Equivalent Type Statement:

Type=UTGRP i=EN,D o=Q

Description:

The UTGRP primitive is identical to the BTGRP primitive, but only provides unidirectional flow. The circuit compiler automatically converts BTGRP elements to UTGRP elements where possible, to improve simulation throughput.

Appendix B SNL Statements and Keywords

Statement Classification

SNL statements can be classified as one of the following:

1. **Annotation**. This can be either text following a **`!DOCUMENTATION (!DOC)`** directive, or text in a single line prefixed by the **`COMMENT (C)`** or **`REMARK (R)`** keywords.
2. **Declare**. This statement begins with a **`%DECLARE (%DCL)`** directive. It specifies attributes, such as print format, for groups of signals.
3. **Delay**. These are statements following a **`!DELAY (!DEL)`** directive. They define global delay-vs.-loading curves, and the correspondence between simulation time-units and real-time.
4. **Format**. This statement begins with a **`!FORMAT (!F)`** directive. It specifies an expected keyword-field ordering for subsequent statements in the Delay and Logical sections. It is used to reduce file size and make files more human readable.
5. **Include**. This statement begins with a **`!INCLUDE (!INC)`** directive. It specifies a list of Network Description files to include during compilation.
6. **Logical**. These are statements following a **`!LOGICAL (!L)`** directive. They describe topological and electrical characteristics of a circuit or subcircuit.

Directives must start a SNL statement (optionally following whitespace). The **`!DOCUMENTATION`**, **`!DELAY`**, and **`!LOGICAL`** directives respectively declare the beginning of sections of statements that contain commentary, delay definitions, and circuit description. They are “sticky”, that is, they remain in effect until one of the other two directives overrides them, or until the end of the Network Description file is reached. For example:

```
!LOGICAL T=buf1 I=a O=b
!LOGICAL P=b T=and I=a O=b
!DELAY DELAY=del1 RISE=10 FALL=12
```

and

```
!LOGICAL
T=buf1 I=a O=b
P=b T=and I=a O=b
!DELAY
DELAY=del1 RISE=10 FALL=12
```

are functionally equivalent. New files are always assumed to start with the **`!LOGICAL`** section. In the above example, the **`!LOGICAL`** directive may be omitted without modifying any meaning, if this is the first text in the file.

Annotation

Annotation can be performed in a number of ways:

1. Beginning the annotating text with a **!DOCUMENTATION** directive. This method is very useful when the text spans many lines. Any text following a **!DOCUMENTATION** directive is, for the most part, ignored by the SIMIC parser. CAUTION: the first non-whitespace character in a line of text should never be '!' or '%', since it may be misinterpreted as a directive.
2. Prefixing the annotation with **COMMENT= (C=)**. This causes the rest of the physical line to be treated as commentary. This type of annotation does not continue beyond a physical line. It is used for in-line comments and small blocks of commentary.
3. Prefixing the annotation with **REMARK= (R=)**. This is similar to **COMMENT**, but if the **REMARK** is placed within an active type block (one that has been instantiated into the circuit being compiled), the annotation will be displayed during the circuit compilation process.
4. Placing commentary to the right of the continuation character (**$**). This allows intra-statement annotation. The **$** must be followed by an equals sign (**=**), otherwise SIMIC will report that it found text beyond a continuation character.

Declare Statements

The **%DECLARE** statements serve two primary functions:

1. Declare statements group signals into an array (vector) under a **root** name within a type block and assign a print radix (format) to the array. Declaring vectors allows groups of signals to be interconnected as a single entity and be displayed in a more convenient format during simulation. The syntax of this declaration for one dimensional arrays is:

   ```
   %DECLARE <format>=<root>[<range>]
   ```

   where ***<range>*** specifies the starting (leftmost) and ending (rightmost) limits of the array. The syntax for two dimensional arrays is:

   ```
   %DECLARE <format>=<root>[<range2>][<range1>]
   ```

   where the ***<range2>*** defines the limits of the second dimension and ***<range1>*** defines limits of the array's first dimension. Each range specification is of the form:

   ```
   <start>:<end>
   ```

   or

   ```
   <start>
   ```

where ***<start>*** and ***<end>*** are integers. In the second form, ***<end>*** is assumed to be equivalent to ***<start>***.

Finally, ***<format>*** is one of the following:

- LEVEL (LEV) – individual levels for each bit.
- OCTAL (OCT) – octal representation.
- HEXADECIMAL (HEX) – hexadecimal representation.
- INTEGER1 (INT1) – One's complement representation.
- INTEGER2 (INT) – Two's complement representation.
- POSINTEGER (POSINT) – Positive integer representation.

The OCT (HEX) print format consists of right-justified groups of 3 (4) bits. The INT1, INT, and POSINT formats consists of a single group containing all array bits. If all bits of a group are tristated (Z), then the single character 'Z' is displayed for the group. Otherwise, if any bit of a group is tristated or 'X' then the single character 'X' is displayed. The following table illustrates sample outputs for a 5 bit vector:

LEV	OCT	HEX	INT1	INT	POSINT
00000	00	00	0	0	0
11111	37	1F	0	–1	31
1000X	2X	1X	X	X	X
ZZZZZ	ZZ	ZZ	Z	Z	Z
ZZZ000	Z0	ZX	X	X	X

2. Declare statements specify a default delay tolerance for delays within a type block. Delay tolerances are percentages that are used to compute minimum and maximum delays from typical delays. This format of the declare statement is:

```
%DECLARE TOLERANCE=<min_max>
```

or

```
%DECLARE TOLERANCE=<min>,<max>
```

where:

<min_max> is a number (fixed or floating point), optionally followed by a percent sign (%), representing the default spread of minimum and maximum delays for elements in this type block, unless explicitly overridden by a delay specification. The minimum delay is computed as

(***<typical delay>*** × (1 - (***<min_max>*** / 100)))

and the maximum delay is computed as

(***<typical delay>*** × (1 + (***<min_max>*** / 100))).

<min> is a number, optionally followed by a percent sign (%), representing the default minimum delay tolerance for the elements in this type block, unless explicitly overridden by a delay specification. The minimum delay is computed as

(***<typical delay>*** $\times$ (1 - (***<min>*** / 100))).

<max> is a number, optionally followed by a percent sign (%), representing the default maximum delay tolerance for the elements in this type block, unless explicitly overridden by a delay specification. The maximum delay is computed as

(***<typical delay>*** $\times$ (1 + (***<max>*** / 100))).

Some examples:

```
%DECLARE TOLERANCE=50%
```

implies: minimum=(typical $\times$ 0.5), maximum=(typical $\times$ 1.5)

```
%DECLARE TOLERANCE=20,50
```

implies: minimum=(typical $\times$ 0.8), maximum=(typical $\times$ 1.5)

```
%DECLARE TOLERANCE=,50
```

implies: minimum=(typical $\times$ 1), maximum=(typical $\times$ 1.5)

```
%DECLARE TOLERANCE=50,
```

implies: minimum=(typical $\times$ 0.5), maximum=(typical $\times$ 1)

If the minimum tolerance is greater than or equal to 100%, then the minimum delay will be set to 0.

Delay Statements

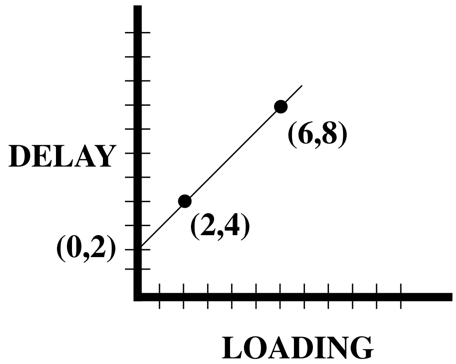
Delay statements specify global delay characteristics and optionally, the relationship between simulation time-units and real-time. They are placed in the **!DELAY** section. If a signal's delay is unspecified, the delay defaults to 0.

The global delay characteristics consist of named delay vs. loading curve families. These delays are referenced by-name in PART and TYPE statements. The delay and loading values defining these delay characteristics may either be fixed-point or floating-point numbers, where a floating point number is represented in exponential notation as a fixed-point number followed by either of the letters ‘E’ or ‘e’ and a positive or negative integral power of 10. For example:

```
37, 370e-1, 0.37e2, 3.7e+1
```

all represent the number 37. The plus sign is optional for positive exponents.

Delay vs. loading curves can be specified either by a two-point form or an intercept-slope form. For example, consider the following delay vs. loading curve:



This characteristic can be described as either:

1. A line that goes through the coordinates (2,4) and (6,8), or,
2. A line that has a y-intercept of 2 and a slope of 1.

The two-point form is:

```
(<load1>,<delay1>)(<load2>,<delay2>)
```

In this example, the two-point representation would therefore be:

```
(2,4)(6,8)
```

The intercept-slope form is:

```
[<intercept>,<slope>]
```

In this example, the intercept-slope representation would be:

```
[2,1]
```

Two-point and intercept-slope forms may be used interchangeably.

If a delay does not vary with loading (the curve has a slope of 0), then only the constant value need be specified. This is the single-point form. For example, the following “curves” are equivalent:

```
(1,4)(10,4)
4
[4,0]
```

Global symbolic names are assigned to each family of delay curves (typical, minimum, and maximum). Delays are then assigned pins or nets by referencing the symbolic names. The format for defining a symbolic delay family is:

```
DELAY=<name> RISE=<curves> FALL=<curves>
```

or

```
DELAY=<name> CHANGE=<curves>
```

In the first form, the rise delays are specified in the **RISE** keyword-field, and the fall delays are specified in the **FALL** keyword-field. If either of these fields is omitted, then the corresponding delays default to 0. In the second form, which may be used when the rise and fall delays are equal, the **CHANGE** keyword-field specifies the common rise and fall delays.

<name> is a user-defined symbolic name to attach to the curves described in the statement, and ***<curves>*** are typical, minimum, and maximum delay curves in the two-point, intercept-slope, and/or single-point formats described above. If the minimum or maximum values are omitted, then they default to the typical value. The components of the family are separated by semicolons (;) as follows:

```
<typical> ; <minimum> ; <maximum>
```

For example, the following delay statement defines a delay whose minimum value is 2, typical value is 3, and maximum value is 4, and assign this delay characteristic the symbolic name **DEMO_DEL**:

```
DELAY=demo_del CHANGE=3;2;4
```

Global delays are assigned to pins of parts and types with the **ODEL** and **BDEL** keywords:

```
part=c type=and i=a,b odel=demo_del
```

In the compilation process, SIMIC totals all pin loading and net loading on each signal. The signal's assigned delay characteristics are then used to compute its delay. Internally, these operations are all performed using floating point arithmetic. The computed value is rounded to the nearest integer.

The relationship between simulation time-units and real-time can be described with the **TIME-UNITS** (**TUNIT**) keyword-field of the **!DELAY** statement. For example, if each SIMIC time-unit corresponds to 1 nS, this statement would be:

```
!DELAY TIME-UNITS=1e-9
```

If specified, SIMIC will display this correspondence in the simulation output header in the PRINT, WRITE, and TGEN output, as well as write it to the history file header information block.

Format Statements

The **!FORMAT** statement reduces the amount of text that needs to be entered. It specifies the expected order of keyword-fields in subsequent SNL statements of a **!DELAY** or **!LOGICAL** section. Having made this specification, all keywords may be omitted in subsequent statements that have this exact format. Only keywords in statements that are exceptions from the expected order need be entered.

The **!FORMAT** statement syntax is:

```
!FORMAT <keyword>= <keyword>= ....
```

where ***<keyword>*** is one of the valid SNL keywords for that section. For example, in the **!DELAY** section, a valid **!FORMAT** statement would be:

```
!FORMAT DELAY= RISE= FALL=
```

This **!FORMAT** statement specifies that subsequent delay statements will contain the delay name followed by the rise specification, followed by the fall specification. A possible format for PART statements in the **!LOGICAL** section, would be:

```
!FORMAT PART= TYPE= I= O=
```

The following are guidelines for using the **!FORMAT** directive:

1. A **!FORMAT** statement may be defined uniquely for the **!DELAY** and **!LOGICAL** sections.
2. A **!FORMAT** statement remains in effect for a section until the next **!FORMAT** statement in that section. This is true even when multiple Network Description Files are used. **!FORMAT** statements remain in effect from file to file.
3. To cancel the **!FORMAT** statement, issue a **!FORMAT** statement without any keywords following it.
4. If a statement does not require a value for one of the keywords specified in the current **!FORMAT** statement, enter a hyphen (**–**) as a "place-holder" for that keyword's value to skip by the keyword.
5. If no values are required for keywords at the end of the **!FORMAT** list, hyphens are not required, since no keywords are skipped.
6. In contrast, if a statement requires a keyword-field that is not specified by the current **!FORMAT** statement, or if you want to specify the keywords in a different order, you must enter the entire keyword-field. This will suspend the formatting for the rest of the statement. Therefore, all subsequent keyword-fields must be explicitly entered (of course abbreviations are acceptable). Clearly, the most efficient way to add a keyword-field is to append it to the end of the line.

The following example demonstrates the various usages of formatted PART statements. In this example, all the PART statements are equivalent:

```
!format   part=     type=     i=        o=
          -         and       a,b part=c
          c         and       a,b
          c         and       i=a,b     o=c
          part=c type=and i=a,b o=c
```

The first PART statement illustrates guidelines (4) and (6). The hyphen skips over the **!FORMAT** statement's **PART=** keyword, so its **AND** entry is associated with the **!FORMAT** statement's **TYPE=** keyword. The entry after the input signal field, (**A,B**), would normally associate with the **!FORMAT** statement's **O=** field, but the **PART=C** entry, containing a keyword, overrides formatting.

The second PART statement illustrates guideline (5). Here, the **O=** entry was omitted (since the output signal's name will default to the identical part name), but no placeholder hyphen was necessary because no other fields follow.

The third PART statement illustrates guideline (6). Having used a keyword to specify the inputs (**I=A,B**), formatting is suspended, and a complete keyword-field (**O=C**) is required to specify the output signal.

Include Statements

Additional files may be referenced from a network description with the **!INCLUDE** directive. The format is:

    !INCLUDE FILE=<*file list*>

where ***<file list>*** is a list of files separated by commas. A default of **FILE=** is defined for this statement, so the statement may be written:

    !INCLUDE <*file list*>

Included files may also contain **!INCLUDE** statements. The current maximum **!INCLUDE** nesting depth of 4.

The contents of an included file are logically placed at the location of the **!INCLUDE** statement referencing the file. Since type blocks are not explicit ended, **!INCLUDE** statements should be carefully placed to avoid inadvertent addition of the included text within a type block definition.

Logical Statements

Logical statements describe circuit topology and electrical characteristics. They are contained in **!LOGICAL** sections, either following a **!LOGICAL** statement or at the beginning of a new file (new files are always assumed to start with a **!LOGICAL** section). There are two basic statements; the TYPE statement and the PART statement.

A TYPE statement begins the definition of a **type block**. It assigns the macro a global type name and defines its external pins. It may also assign electrical attributes to the pins and declare the type block's level of abstraction (composition).

In a structural type block, or macro, the TYPE statement is followed by one or more PART statements that instantiate components within the type block.

PART statements may also assign electrical attributes to the pins of instantiated components.

A type block ends either when a new TYPE statement, or the end of all files, has been reached during circuit compilation.

While TYPE statements and PART statements perform totally different functions (*semantic* differences), they only differ *syntactically* by the presence of a PART keyword-field, which must appear in PART statements, but cannot exist in TYPE statements. This PART field specifies an instance name for the referenced type. Both statements must contain a TYPE keyword-field and at least one of the following keyword-fields: I (inputs), O (outputs), or B (bidirectional).

SNL supports many keywords for PART and TYPE statements. These keywords can be classified as follows:

1. **Topological**. These include the keywords: PART, TYPE, I, O, B and COMP.
2. **Electrical attributes**. These specify delay and decay curves (ODEL, BRISE, ODEC, etc.), loading (ILOD, OLOD, BLOD), wire-tie operations (ODOM, BDOM), spike control (OFILT, OLIB, BFILT, BLIB), high impedance default (IHIZ), etc.
3. **Physical attributes**. These include: number of pads (PADS), number of transistors (TRANS), cell widths (W); which are totaled and displayed after circuit loading. Also attributes such as pad placement (APAD, IPAD, OPAD, BPAD) and power pads (A) are ignored by SIMIC, but are supported as valid keywords to allow use of SNL descriptions for other applications (e.g. place and route programs).
4. **Special characteristics**. These attribute apply only to a class of primitive types. The include timing checks (TIMING-CHECKS), ON-resistance for switches (SDEPTH), and last valid address for ROMS and RAMS (LASTADDR).

Electrical attributes are assigned to pins <u>in their respective order</u> for TYPE statements, and to nets <u>in their respective order of connection</u> for PART statements.

Example: Assigning Electrical Attributes

Suppose a cell library contains a three-input, three output cell named **`6370`** with:

- input pins named **`ia`**, **`ib`**, and **`ic`** having pin loads of 1, 2, and 3 respectively, and
- output pins named **`ox`**, **`oy`**, and **`oz`** having the electrical attributes:
  `ox`: pin-load=10, rise-delay=20, fall-delay=0
  `oy`: pin-load=9, delays defined by global delay **`del13`**

oz: pin-load=8, delays defined by global delay **del50**

The cell's TYPE statement might be:

```
TYPE=6370 I=ia,ib,ic O=ox,oy,oz ILOD=1,2,3 $
    OLOD=10,9,8 ODEL=,del13,del50 ORISE=20
```

If an instance of this cell is named **q**, and has input signals named **a**, **b**, and **c** and output signals named **x**, **y**, and **z**, then the PART statement:

```
PART=q TYPE=6370 I=a,b,c O=x,y,z
```

places this cell in the circuit and causes this instance to inherit all electrical attributes of the cell library definition (that is, loads of 1, 2, and 3 are added to nets **a**, **b**, and **c**, respectively, signal **x** has a rise-delay of 20 and a fall-delay of 0, etc.).

If this is a custom library rather than a standard cell library, then different instances of the same cell may have different electrical attributes. For example, suppose that instance **q** of the **6370** differs from the cell definition in the following manner: (a) the loading at pins **ia**, **ic**, and **ox** are 5, 6, and 12, respectively, and (b) the delay characteristics of pins **ox** and **oy** are described by global delays **del30** and **del40**, respectively. If connections are made **by-order**, then the instantiating PART statement might be:

```
PART=q TYPE=6370 I=a,b,c O=x,y,z ILOD=5,,6 $
    OLOD=12 ODEL=del30,del40
```

Alternatively, if connections are made by-pin-name, then the instantiating PART statement might be:

```
PART=q TYPE=6370 I=c(ic),b(ib),a(ia) $
    O=z(oz),y(oy),x(ox) ILOD=6,,5 $
    OLOD=,,12 ODEL=,del40,del30
```

End of Example

Supported PART and TYPE Statement Keywords

The following table summarizes all currently-supported SNL keywords for PART and TYPE statements. For each keyword, it (a) lists all keyword aliases, (b) briefly describes the keyword's function, (c) describes the expected value's form (i.e., the type of entry expected on the right hand side of the keyword's equal sign), and (d) where applicable, the default value if the keyword is omitted.

Most keywords, or their minimum abbreviations, are valid in both PART and TYPE statements. Those keywords that are only valid for TYPE statements are marked with "(T)", and those that are only valid for PART statements are marked with "(P)".

The expected keyword values fall into the following categories:

1. <u>Delay curve format</u> – the format for describing delay curve families, as described in the Delay Statements section of this Appendix,
2. <u>Global delay reference</u> – the name of a delay relation, as specified in the delay statement's **`DELAY=`** keyword-field,
3. <u>A numeric value</u> – either an integer, or a "number" which, in this table, represents either a fixed-point or floating-point numeric format (note that an integer is a special case of the fixed-point format),
4. <u>A reserved value</u> – represented in the table by capitalized words (e.g., INFINITE, POWER, etc.). Any valid prefix of these reserved words may be entered,
5. Signal name format – A valid signal name, which is either
   a) any sequence of characters beginning with an underscore (_), question mark (?) or an alphanumeric character (0-9, a-z, A-Z) and optionally containing more characters that are alphanumeric characters, underscores, question marks, hyphens, percent signs, exclamation marks, and periods, or
   b) any sequence of characters enclosed in single or double quotes.
6. <u>Specialized specification blocks</u> – BOOLEAN and TIMING-CHECKS blocks.

.

Keyword	Description
A APIN AUX-PINS	Specifies auxiliary pad pins such as VDD and VSS. Although not used by SIMIC, it is valuable for interfacing to place and route programs. *Value*: Currently undefined.
APAD AUX-PADS	Specifies location to place the corresponding auxiliary pad pin (APIN) for place and route programs. *Value*: Currently undefined.
B BPIN (T) BNET (P) BUS-PINS (T) BUS-NETS (P)	Specifies a bidirectional pin (T) or signal (P) name. *Value*: Signal name format.
BCHANGE BUS-CHANGE	Local delay specification for equal rise and fall delays at a bus. *Value*: Delay curve format. *Default*: 0 rise and fall delays.
BDEC BUS-DECAY	Specifies decay time for a bus. *Value*: Global delay reference, a number, or INFINITE. *Default*: 0 (instantaneous) decay.
BDEL BUS-DELAY	Specifies bus rise and fall delays using global delay statements. *Value*: Global delay reference. *Default*: 0 rise and fall delays.
BDOM BUS-DOMINANCE	Specifies wire-tie dominance for a bus. *Value*: 0, 1, or X (for wired-AND, wired-OR, or No dominance, respectively). *Default*: X (No dominance).
BDRIVE BUS-DRIVE	Specifies both high and low drive strengths for a bus. *Value*: POWER, DRIVING, RESISTIVE, FLOATING. *Default*: DRIVING.

Keyword	Description
BFALL BUS-FALL	Local fall delay specification for a bus. *Value*: Delay curve format. *Default*: 0 fall delays.
BFILTER BUS-FILTER	Specifies spike filter parameter for a bus. *Value*: Integer, 0 through 100, optionally followed by percent (%) sign. *Default*: 0.
BHDRIVE BUS-HDRIVE	Specifies high drive strength for a bus. *Value*: POWER, DRIVING, RESISTIVE, FLOATING. *Default*: DRIVING.
BLDRIVE BUS-LDRIVE	Specifies low drive strength for a bus. *Value*: POWER, DRIVING, RESISTIVE, FLOATING. *Default*: DRIVING.
BLIBERAL BUS-LIBERAL	Specifies spike liberal parameter for a bus. *Value*: Integer, 0 through 100, optionally followed by percent (%) sign. *Default*: 0.
BLOD BUS-LOADS	Specifies loading at a bus pin. *Value*: Number *Default*: 0.
BOOLEAN	For BOOLEAN part and type specification of BOOLEAN equations. Invalid in any other case. *Value*: BOOLEAN equation block.
BPAD BUS-PADS	Specifies location of bidirectional pads for place and route programs. *Value*: Signal name format.
BRISE BUS-RISE	Local rise delay specification for a bus. *Value*: Delay curve format. *Default*: 0 fall delays.
C COM COMMENT	Comment to end of physical line.

Keyword	Description
COMP COMPOSITION	Specifies the composition of the type being defined in a TYPE statement. *Value*: MACRO, BOOLEAN, BEHAVIORAL. *Default*: MACRO. Specifies the referenced type's composition. in a PART statement. *Value*: MACRO, PRIMITIVE, BOOLEAN, BEHAVIORAL. *Default search order*: MACRO, BOOLEAN, BEHAVIORAL, PRIMITIVE
I INET (P) IPIN (T) INPUT-NETS (P) INPUT-PINS (T)	Specifies an input pin (T) or signal (P) name. *Value*: Signal name format.
IHIZ INPUT-HIZ	Specifies how to treat a floating input to a gate-level component. *Value*: 0, 1, or X. *Default*: X.
ILOD INPUT-LOADS	Specifies loading at an input pin. *Value*: Number *Default*: 0.
IPAD INPUT-PADS	Specifies location of input pads for place and route programs. *Value*: Signal name format.
LASTADDR	For RAMS and ROMS, specifies the last valid address. *Value*: Integer. *Default*: Highest addressable location.
O ONET (P) OPIN (T) OUTPUT-NETS (P) OUTPUT-PINS (T)	Specifies an output pin (T) or signal (P) name. *Value*: Signal name format.
OCHANGE OUTPUT-CHANGE	Local delay specification for equal rise and fall delays at an output. *Value*: Delay curve format. *Default*: 0 rise and fall delays.

Keyword	Description
ODEC OUTPUT-DECAY	Specifies decay time for an output. *Value*: Global delay reference, number, or INFINITE. *Default*: 0 (instantaneous) decay.
ODEL OUTPUT-DELAY	Specifies rise and fall delays for an output using global delay statements. *Value*: Global delay reference. *Default*: 0 rise and fall delays.
ODOM OUTPUT-DOMINANCE	Specifies wire-tie dominance for an output. *Value*: 0, 1, or X, for wired-AND, wired-OR, or No dominance, respectively. *Default*: X (No dominance).
ODRIVE OUTPUT-DRIVE	Specifies both high and low drive strengths for an output. *Value*: POWER, DRIVING, RESISTIVE, FLOATING. *Default*: DRIVING.
OFALL OUTPUT-FALL	Local fall delay specification for an output. *Value*: Delay curve format. *Default*: 0 fall delay.
OFILTER OUTPUT-FILTER	Specifies spike filter parameter for an output. *Value*: Integer, 0 through 100, optionally followed by percent (%) sign. *Default*: 0.
OHDRIVE OUTPUT-HDRIVE	Specifies high drive strength for an output. *Value*: POWER, DRIVING, RESISTIVE, FLOATING. *Default*: DRIVING.
OLDRIVE OUTPUT-LDRIVE	Specifies low drive strength for an output. *Value*: POWER, DRIVING, RESISTIVE, FLOATING. *Default*: DRIVING.

Keyword	Description
OLIBERAL OUTPUT-LIBERAL	Specifies spike liberal parameter for an output. *Value*: Integer, 0 through 100, optionally followed by percent (%) sign. *Default*: 0.
OLOD OUTPUT-LOADS	Specifies loading at an output pin. *Value*: Number *Default*: 0.
OPAD OUTPUT-PADS	Specifies location of output pads for place and route programs. *Value*: Signal name format.
ORISE OUTPUT-RISE	Local rise delay specification for an output. *Value*: Delay curve format. *Default*: 0 rise delay.
P PART	Specifies an instantiation name for a referenced TYPE in a macro definition (type block). *Value*: Signal name format.
PADS	Specifies the number of physical pads for the cell (macro). Values are summed for each occurrence of the cell. Used by place and route programs. *Value*: Integer. *Default*: 0
R REM REMARK	Comment to end of physical line. Remark is displayed during circuit compilation.
SDEPTH SERIES-DEPTH	For resistive switches, specifies the “ON resistance” of the switch. *Value*: Integer from 1 to 32766. *Default*: 1
STATE	For BOOLEAN types and parts, specifies the BOOLEAN state variables for that type or part. *Value*: Signal name format. *Default*: No state variables.

Keyword	Description
T TYPE	Defines a macro name in a TYPE state-ment. References a TYPE in a PART statement. *Value*: Signal name format.
TIMING-CHECKS	D-latch and all flip-flops, specifies timing check values, such as SETUP and HOLD. *Value*: TIMING-CHECKS block. *Default*: All timing checks disabled.
TRANS TRANSISTORS	Specifies the number of transistors for a defined cell (macro). Values are summed for each instantiation of the cell. Used by place and route programs. *Value*: Integer. *Default*: 0
W WIDTH	Specifies the cell width for standard cells, or number of gates for gate array designs. Used by place and route programs. *Value*: Integer. *Default*: 0

Appendix C Run Commands and Keywords

Overview

This Appendix summarizes the keyword options and expected values for SIMIC run commands. The appropriate sections of this Guide should be consulted for complete descriptions of these options.

Comments may be placed at the end of any run command or between commands. Each comment begins with the **`COMMENT`** (**`C`**) keyword, followed by an equal sign (=). Comments always end at the end of the physical line. Remarks may also be placed at the end of run commands or between them. Remarks are similar to comments, except that they are written to the **tester interface file**. Each remark begins with the **`REMARK`** (**`REM`**) keyword, followed by an equal sign (=).

The supported keyword values for two common keywords, **`LIST`** and **`PRANGE`**, are described below.

Table I lists the supported SIMIC commands and their minimum abbreviations.

Table II lists the supported SIMIC keywords and their minimum abbreviations.

Table III lists the supported SIMIC reserved keyword values recognized for certain keyword fields, and their minimum abbreviations.

SIMIC recognizes any valid abbreviation of reserved words in Tables I through III, ranging from the given minimum abbreviation to the full word.

Table IV summarizes expected keyword values; these values are described under the commands that require them. Many keywords are used by multiple commands.

The LIST Keyword

The expected value for **`LIST`** keyword is designated as ***<signals>*** in this Appendix. Other keywords (for example, **`ONE`** and **`ZERO`**) also accept the same specification. The following can be included in the list of specifications for ***<signals>***:

1. Signal names
2. Meta-words specifying primary signal values:
    a. **&INPUTS** – specifies “all primary inputs”
    b. **&OUTPUTS** – specifies “all primary inputs”

c. **&BUSSES** – specifies “all primary busses”

d. **&BUSINS** – specifies “the stimulus values at all primary busses” which may differ from the values of **`&BUSSES`** (the latter being the wire-tied result of primary stimulus values and internally-driven values)

3. Wildcard specification of the form ***<prefix>*()**, which represents “all signal names beginning with ***<prefix>***” (for example, **`A.B.()`** specifies “all signals whose names begin with **`A.B.`**”)
4. Factored specification of the form ***<prefix>(<suffix1>, …, <suffixn>)***, which represents the signals
 <prefix><suffix1>, …, <prefix><suffixn>
(for example, **ab(cd, ef, gh)** *specifies* **abcd***,* **abef***, and* **abgh***)*
5. **Vector aliases** defined in a **`DEFINE`** command (see the *Specifying Signal Groups and Output Radix Format* Section in this Chapter)
6. Alternative specification of each signal in the form ***<partname>.<n>***, where ***<partname>*** is the instance name of the part generating the signal, and ***<n>*** is the signal’s output number (e.g., if the signal is the fourth output of this element, then ***<n>*** would be 4*)*.

The PRANGE Keyword

Each command that supports the **`PRANGE`** keyword maintains its own separate specification (i.e., a **`PRANGE`** specification for one command does not affect the active interval of any other command).

The **`PRANGE`** keyword values are test intervals for pattern and timing generator stimuli, or time intervals for waveform stimuli. Integers are always used in this specification.

If no **`PRANGE`** option is specified for a command, then SIMIC defaults the command’s active interval to “enabled for the entire simulation range” (equivalent to **`PRANGE:`**)

A **`PRANGE`** interval is specified as:

<starting point>–<ending point>

For example, the following command restricts output to the **write** file from test (time) 100 to test (time) 200 only:

```
WRITE PRANGE=100-200
```

`PRANGE` specifications are “sticky”, or cumulative. For example, if the command:

```
WRITE PRANGE=300-400
```

were issued after the above command, the combined effect would be identical to the single command:

```
WRITE PRANGE=100-200, 300-400
```

As a special case of an interval, if ***<starting point>*** is omitted, 0 is assumed. Thus,

```
WRITE PRANGE=-20000
```

enables the **`WRITE`** command from 0 (beginning of the simulation) through test (or time) 20,000.

<ending point> may also be omitted in the interval specification. In this case, the interval is open-ended (until the last input stimulus state has been propagated). For example:

```
WRITE PRANGE=30000-
```

enables the **`WRITE`** command from test (time) 30,000 to the end of simulation.

Individual tests (particular times) are specified as a single integer with no hyphen.

When a **`NO`** prefix is used with a command, it excludes the **`PRANGE`** interval. For example, the sequence of commands:

```
WRITE PRANGE=1000-2000
NO WRITE PRANGE=1101-1199
```

enables **write** output from test (time) 1000 to 2000, but disable this output from test (time) 1101 to 1199. The two commands are equivalent to the single command:

```
WRITE PRANGE=1000-1100,1200-2000
```

Table I Commands And Their Minimum Abbreviations

APPLY (AP)	?LOADING (?LO)
BREAK (BR)	?PRINT (?PR)
CLAMP (CL)	?SPIKES (?SP)
DEFINE (DE)	?WRITE (?WR)
EXECUTE (EX)	QUIT (QU)
GET (GE)	RESTORE (RE)
HISTORY (HI)	SAVE (SA)
LOOK (LO)	SET (SE)
PRINT (PR)	SIMULATE (SI)
?CHECKS (?CH)	TGEN (TG)
?DECAY (?DEC)	TRACE (TR)
?DEFINE (?DE)	WARN (WA)
?DELAY (?DEL)	WRITE (WR)
?DRIVE (?DR)	XPROPAGATE (XP)

Table II Keywords And Their Minimum Abbreviations

AFILE (AF)	OSCILLATION (OS)
AND (AN)	OUTPUTS (OU)
BEGIN (BE)	PARTS (PAR)
BITMAP (BI)	PATTERNS (PA)
BOOLEAN (BO)	PERIOD (PE)
BTG (BT)	PLA (PL)
BTGCONTROLS (BTGC)	PRANGES (PR)
BTGDELAY (BTGD)	PSTEP (PS)
BTGSOURCES (BTGS)	PULSE (PU)
CHANGE (CH)	RDEPTHS (RD)
COMMENT (C)	REMARK (REM)
CONFLICT (CON)	REPORT (REP)
DATA (DA)	RFILE (RF)
DECAY (DE)	RISE (RI)
DISCONNECT (DI)	ROM (RO)
ENABLE (EN)	SFILE (SF)
EXPAND (EX)	SPIKE (SP)
FALL (FA)	STABILITY (STAB)
FILE (FI)	STOP (STOP)
FILTER (FILT)	STRING (STRI)
HAZARD (HA)	STROBE (STRO)
HEADER (HE)	TARGET (TA)
HIZ (HI)	TERM (TE)
INPUTS (IN)	TIMING (TI)
LFILE (LF)	TNUM (TN)
LIBERAL (LIB)	TSTEP (TS)
LIST (LI)	TYPE (TY)
MEMLATCH (MEML)	UNSTABLE (UN)
MEMSPIKE (MEMS)	VALUES (VA)
NEAR (NE)	X (X)
ONE (ON)	XADDRESS (XA)
OR (OR)	ZERO (ZE)

Table III Keyword Values And Their Minimum Abbreviations

&ALPHANUMERIC (&A)	LEVELS (L)
&NUMERIC (&N)	MAXIMUM (MA)
ALL (A)	MINIMUM (MI)
BINARY (B)	OCTAL (O)
DRIVING (D)	POSINTEGER (P)
DYNAMIC (D)	POSTDECAY (PO)
FLOATING (F)	POWER (P)
HEXADECIMAL (H)	PREDECAY (PR)
INFINITE (I)	RESISTIVE (R)
INT1 (INT1)	STATIC (S)
INTEGER (I)	STRENGTHS (S)
INTEGER1 (INTEGER1)	SYMBOLS (S)
INTEGER2 (I)	TYPICAL (T)
LEVEL (L)	

Table IV Syntactic Items

Syntactic Item	Description
<address_data>	RAM content specification consisting of a sequence of segments, where each segment contains an initial address—the letter X followed by the hexadecimal address—followed by the segment's hexadecimal data.
<boolean_block>	One or more semicolon-terminated Boolean equations enclosed within a BEGIN END block.
<connection_map>	A sequence of 1, 0, or X characters representing true, complemented, or no-connection, respectively, to define the AND or OR plane personalities of a PLA. The number of characters must equal the plane size specified with the respective **AND** or **OR** keyword fields.
<defaults>	Optional dot-delimited defaults specification for primitive entries of a defined pattern or waveform, consisting of: • an integer specifying the default duration, • a default format specification; any prefix of BINARY, OCTAL, HEXADECIMAL, or INTEGER, • a default strength specification; any valid prefix of POWER, DRIVING, RESISTIVE, or FLOATING
<decay spec>	Either an integer, an integer percentage, or a prefix of INFINITE. An integer percentage is an integer between 0 and 100, inclusive, followed by a percent sign (%), and optionally preceded by a plus (+) or minus(-) sign.
<delay spec>	Either an integer or an integer percentage.The latter is an integer between 0 and 100, inclusive, followed by a percent sign (%), and optionally preceded by a plus (+) or minus(-) sign.
<dformat>	Timing generator drive mask specification: NRZ, RZ, RO, or RC.

Table IV Syntactic Items

Syntactic Item	Description
<dis_val>	Either 0 or 1, specifying PLA output values when the enable input, EN, is 0.
<dmarks>	Comma-separated list of integers representing drive mask event times (referenced from start of the time-set).
<eformat>	Timing generator enable mask specification: NE, RD, or RF.
<emarks>	Comma-separated list of integers representing enable mask event times (referenced from start of the time-set).
<file name>	A file name consistent wilth the operating system. This specification could optionally contain a pathname or directory as well as the file's extension (to override SIMIC file extension defaults). Names containing characters other than alphanumerics, underscores, hyphens, exclamation marks, question marks, or periods should be enclosed in quotes.
<file list>	Either a single *<file name>*, or a comma-separated list of *<file name>* specifications.
<filt_lib>	An integer between 0 and 100 inclusive, optionally followed by a percent (%) sign.
<filter string>	A pattern of characters, possibly enclosed in quotes, and optionally preceded by an integer repetition factor and asterisk (*).
<format>	Format specification consisting of any prefix of HEXADECIMAL, INTEGER2, LEVEL, OCTAL, or POSINTEGER, or the full words INT1 or INTEGER1.
<integer>	Integer specifying a count or multiplier.
<main type>	The name of the subcircuit to be loaded and simulated.
<name>	The name of a defined stimulus sequence, timing generator, strobe, or vector alias stripped of the initial P, W, T, S, or V letter. The name must contain valid SIMIC name characters.

Table IV Syntactic Items

Syntactic Item	Description
<ninputs>	Integer specifying the number of inputs to the AND plane of a PLA.
<noutputs>	Integer specifying the number of outputs of the OR plane of a PLA.
<nproducts>	Integer specifying the number of outputs of the AND plane of a PLA.
<part_bool>	The hierarchical name of a BOOLEAN instance.
<part_pla>	The hierarchical name of a PLA instance.
<part_ram>	The hierarchical name of a RAM instance.
<parts_ff>	Comma-separated list of SIMIC latch or flip-flop instance (part) names.
<pname>	The name of a pattern stimulus sequence, beginning with the letter P.
<prange spec>	A comma-separated specification of integer ranges and/or integers representing **`PRANGE`** keyword values.
<pw>	Either the letter P or the letter W, respectively designating the start of a pattern or waveform name..
<sequence>	The hierarchical definition of a pattern or waveform sequence consisting of primitive entries, named references, positioning specifications, and loops.
<sformat>	Strobe format; either SP or SW.
<signals>	Comma-separated list of signals, either primary or secondary (internal).
<signals and format>	Comma-separated list of signals, either primary or secondary (internal), and the asterisk (*) and sharp (#) format control characters.
<signals_ff>	Comma-separated list of D-latch or edge-triggered flip-flop outputs (SIMIC primitives).
<signals_pi>	Comma-separated list of primary inputs and/or primary busses.

Table IV Syntactic Items

Syntactic Item	Description
<signals_po>	Comma-separated list of primary outputs and/or primary busses.
<smarks>	Single integer, or comma-separated pair of integers representing strobe event times (referenced from start of the time-set).
<strobe>	The name of a strobe beginning with the letter S.
<table>	The name of a timing table, a prefix of TYPICAL, MINIMUM, or MAXIMUM.
<time>	Integer specifying a simulation time (time-units).
<time-set>	The name of a time-set, beginning with the letter T.
<timing-check>	The name of a supported timing check, either qualified (e.g., SETUP.D, HOLD.K, PW.C.L) or unqualified (e.g., SETUP, HOLD, PW). Timing check names may not be abbreviated. See Appendix A for descriptions of the timing checks associated with SIMIC latch and edge-triggered flip-flop primitives.
<tnum>	Integer specifying a test number.
<type_bool>	The name of a BOOLEAN TYPE.
<width>	Integer specifying the width of a pattern or waveform.
<wname>	The name of a waveform stimulus sequence, beginning with the letter W.

Run Command: APPLY (AP)

Function

The **APPLY** command associates named stimulus sequences—patterns or waveforms, previously specified with one or more **DEFINE** commands—with primary inputs (including primary busses). All primary inputs must be assigned stimuli before simulation can begin. The **APPLY** commands may specify either patterns or waveforms, but not both.

For Tester Emulation Mode, the **APPLY** command additionally:

1. associates named time-sets—timing generator characteristics previously specified with **DEFINE** commands—with primary inputs
2. associates named output strobes—previously specified with **DEFINE** commands—with primary outputs
3. changes master test period.

Usage

Associating Stimuli With Primary Inputs And Busses

To associate a pattern that starts at test number <*tnum*> with primary inputs and/or primary busses:

```
APPLY PATTERNS=<pname> LIST=<signals_pi> $
    BEGIN=<tnum>
```

The number of primary inputs specified in the LIST keyword-field must be equal to the width of pattern <*pname*>.

If the **BEGIN** keyword option is omitted, the pattern will start immediately, if at the start of simulation or if the circuit state is stable, or when the circuit state becomes stable. For example:

```
APPLY PATTERNS=pabc LIST=pi1,pi2 BEGIN=101
```

associates the pattern **pabc** (of width 2) with primary inputs **pi1** and **pi2**, and schedules this association to start at Test 101, while

```
APPLY PATTERNS=pabc LIST=pi1,pi2
```

causes the pattern to immediately determine the next values of these signals.

To associate a waveform that starts at time <*time*> with primary inputs and/or primary busses:

```
APPLY PATTERNS=<wname> LIST=<signals_pi> $
    BEGIN=<time>
```

The number of primary inputs specified in the LIST keyword-field must be equal to the width of waveform <*wname*>.

If the **BEGIN** keyword option is omitted, the waveform will start immediately, if at the start of simulation, or one time-unit after the command is issued. For example:

```
APPLY PATTERNS=wabc LIST=pi1,pi2 BEGIN=101
```

associates the waveform **wabc** (of width 2) with primary inputs **pi1** and **pi2**, and schedules this association to start at Time 101, while

```
APPLY PATTERNS=wabc LIST=pi1,pi2
```

causes the waveform to immediately determine the next values of these signals.

The **LIST** keyword may be omitted for the special case that (a) the stimulus width is equal to the number of primary inputs and busses, and (2) bit positions within the stimulus correspond to the ordering of inputs and busses in the main type's TYPE statement, with input values preceding bus values.

Associating Time-Sets With Primary Inputs And Busses

To associate a time-set (timing generator definition) with one or more primary inputs and/or busses starting at Test <*tnum*>:

```
APPLY TIMING=<time-set> LIst=<signals_pi> $
     BEgin=<tnum>
```

For example

```
APPLY TIMING=timing1 LIST=pi1,pi2 BEGIN=150
```

associate the time-set **timing1** with primary inputs **pi1** and **pi2**, starting at Test 150.

If the **BEGIN** keyword option is omitted, the time-set will start immediately, if at the start of simulation or if the circuit state is stable, or when the circuit state becomes stable.

If no time-set has been **APPLY**ed to a primary input or bus, then the default timing generator **tdefault.nrz.ne**=0,0:0,0 (zero rise, fall, tristate, and drive delays) is associated.

Associating Strobes With Primary Outputs And Busses

The **TIMING** and **LIST** keywords are also used to associate a strobe with one or more primary outputs and/or busses starting at Test <*tnum*>:

```
APPLY TIMING=<strobe> LIST=<signals_po> $
     BEGIN=<tnum>
```

For example

```
APPLY TIMING=strob1 LIST=po1,po2 BEGIN=150
```

associate the strobe **strob1** with primary outputs **pi1** and **pi2**, starting at Test 150.

If the **BEGIN** keyword option is omitted, the strobe will start immediately, if at the start of simulation or if the circuit state is stable, or when the circuit state becomes stable.

If no strobe has been **APPLY**ed to a primary input or bus, then a default point (edge) strobe, placed one-time unit prior to the end of the test period, is associated.

Changing The Master Test Period

The master test period can be changed during simulation with the command:

`APPLY PERIOD=<time> BEGIN=<tnum>`

For example:

`APPLY PERIOD=1000 BEGIN=150`

schedules the master test period to change to 1000 time-units at Test 150.

Run Command: BREAK (BR)

Function

The **BREAK** command causes simulation to stop when user-specified conditions occur. At a breakpoint, the state of the simulated circuit is “frozen”, allowing the user to perform interactive debugging (probe signals, change delays, resimulate, etc.) or to control the course of future simulation.

Previously-set **BREAK** conditions are cancelled using the **NO** prefix. Unless explicitly indicated, keyword options that support the ***keyword=value*** field format also support this format with the **NO** prefix. For example:

```
NO BREAK RISE=sig1
```

cancels the specific break when a rise transition occurs at signal **sig1**. All keyword options support the ***keyword:*** field format with the **NO** prefix. For example:

```
NO BREAK RISE:
```

cancels all previous breakpoints on rise transitions.

Usage

See the Section *Setting Simulation Breakpoints* in Chapter 2.6 for a complete description of the **BREAK** command.

The **BREAK** command can be restricted to a specified interval of tests, for patterns, or time, for waveforms, with the **PRANGE** keyword option:

BREAK PRANGE=<*prange spec*>

Break On The Occurrence Of Signal Transitions

The **CHANGE** keyword specifies any level transition; either a rise,0→X, 0→1, X→1, or a fall, 1→X, 1→0, X→0:

BREAK CHANGE=<*signals*>

sets a breakpoint when any of the specified signals changes state, while:

```
BREAK CHANGE:
```

sets a breakpoint when any transition occurs (for single-stepping through the simulation).

The **DECAY** keyword specifies a decay to Z (floating unknown):

BREAK DECAY=<*signals*>

sets a breakpoint when any of the specified signals decays, while:

```
BREAK DECAY:
```

sets a breakpoint when any signal decays.

The **RISE**, **FALL**, **CHANGE**, **X**, and **DECAY** keywords accept the same type of signal specification as the **LIST** keyword.

The **FALL** keyword specifies a fall transition; 1→X, 1→0, X→0:

```
BREAK FALL=<signals>
```

sets a breakpoint when any of the specified signals executes a fall transition, while:

```
BREAK FALL:
```

sets a breakpoint when any signal executes a fall transition.

The **MEMLATCH** keyword specifies a transition from a known state to an unknown state at a SIMIC primitive D-latch or edge-triggered flip-flop due to a sensitized unknown input (e.g., unknown clock):

```
BREAK MEMLATCH=<signals_ff>
```

sets a breakpoint when an unknown input value is sensitized at a built-in memory element whose output is one of the specified signals, while:

```
BREAK MEMLATCH:
```

sets a breakpoint when the state of any built-in memory element becomes unknown due to a sensitized unknown input.

The **RISE** keyword specifies a rise transition; 0→X, 0→1, X→1:

```
BREAK RISE=<signals>
```

sets a breakpoint when any of the specified signals executes a rise transition. For example:

```
BREAK RISE=sig1,sig2
```

will cause a break from simulation any time signals **sig1** or **sig2** execute a transition to logic-1. The command:

```
BREAK RISE:
```

sets a breakpoint when any signal executes a rise transition.

The **X** keyword specifies a level transition from a known state to an unknown state, 1→X, 0→X:

```
BREAK X=<signals>
```

sets a breakpoint when any of the specified signals becomes unknown, while:

```
BREAK X:
```

sets a breakpoint when any signal becomes unknown.

Break On The Occurrence Of Timing Hazards

The timing hazards described in this section are primarily due to the distribution of propagation delays within the circuit.

The **HAZARD** keyword is a shorthand designation of all switching hazards—**pulse**, **spike**, or **near**. The command:

```
BREAK HAZARD=<signals>
```

sets a breakpoint when a pulse, spike, or near hazard occurs at any of the specified signals. For example, the command:

```
BREAK HAZARD=sig1
```

is equivalent to:

```
BREAK PULSE=sig1 SPIKE=sig1 NEAR=sig1
```

The command:

```
BREAK HAZARD:
```

sets a breakpoint when a pulse, spike, or near hazard occurs at any signal.

The **MEMSPIKE** keyword specifies the occurrence of a **spike hazard** at a flip-flop primitive, and is therefore a subset of **SPIKE**. It narrows focus to spikes that could cause steady-state errors. The command:

BREAK MEMSPIKE=<*signals_ff*>

sets a breakpoint when a spike hazard occurs at any of the specified flip-flop output signals, while:

```
BREAK MEMSPIKE:
```

sets a breakpoint when a spike hazard occurs at any flip-flop. When a spike occurs at a flip-flop for which the **MEMSPIKE** keyword option has been specified, the break message will contain *memspike* as the cause of the break, rather than *spike*.

The **NEAR** keyword specifies the occurrence of a **near hazard** (see also **DEFINE NEAR**). The command:

BREAK NEAR=<*signals*>

sets a breakpoint when a near hazard occurs at any of the specified signals, while:

```
BREAK NEAR:
```

sets a breakpoint when a near hazard occurs at any signal.

The **PULSE** keyword specifies the occurrence of a **pulse hazard** (see also **DEFINE PULSE**). The command:

BREAK PULSE=<*signals*>

sets a breakpoint when a pulse hazard occurs at any of the specified signals, while:

```
BREAK PULSE:
```

sets a breakpoint when a pulse hazard occurs at any signal.

The **SPIKE** keyword specifies the occurrence of a **spike hazard**. The command:

BREAK SPIKE=<*signals*>

sets a breakpoint when a spike hazard occurs at any of the specified signals, while:

```
BREAK SPIKE:
```

sets a breakpoint when a spike hazard occurs at any signal.

The **STROBE** keyword specifies the condition that a **strobe error** occurs at any primary output or bus. This is only defined for tester emulation mode. The command:

```
BREAK STROBE:
```

sets a breakpoint for this situation.

The **UNSTABLE** keyword designates the condition that a change of primary input state has occurred while the circuit was still in an unstable state (loss of fundamental mode of operation for waveform and tester emulation modes). The command:

```
BREAK UNSTABLE:
```

sets a breakpoint for this situation.

Break On The Occurrence Of Timing Check Violations

Breakpoints can be set for timing check violations at SIMIC latches and flip-flops. The **PARTS** keyword is used to specify which memory elements to monitor for the selected timing check(s). The command:

BREAK PARTS=<*parts_ff*> <*timing-check*>: ...

selects specific memory elements to monitor. For example,

```
BREAK PARTS=ff1,ff2 SETUP.D: PW:
```

sets breakpoints for data setup-time and all pulse-width check violations at the memory elements named **ff1** and **ff2**.

The command:

BREAK PARTS: <*timing-check*>: ...

sets breakpoints when a violation of the specified timing checks occurs at any memory element. For example,

```
BREAK PARTS: SETUP: HOLD: PW:
```

will cause any timing check violation at any memory element to trigger a break from simulation.

Break On The Occurrence Of Wire-tie Conflicts

The **CONFLICT** keyword specifies a wire-tie conflict. The command:

BREAK CONFLICT=<*signals*>

sets a breakpoint when a wire-tie conflict occurs at any of the specified signals, while the command:

```
BREAK CONFLICT:
```

sets a breakpoint when any wire-tie conflict occurs.

The **CONFLICT** keyword accepts the same type of signal specification as the **LIST** keyword.

Break On Oscillation

The **OSCILLATION** keyword specifies excessive activity in response to a single change of input state (see also **DEFINE OSCILLATION**). The command:

```
BREAK OSCILLATION:
```

sets a breakpoint when any signal exhibits excessive activity.

Break Periodically

The **PSTEP** keyword specifies a stable-state count. The command:

BREAK PSTEP=<*integer*>

causes a break from simulation whenever the circuit state has become stable the specified number of times. For example,

```
BREAK PSTEP=5
```

sets a breakpoint every fifth stable state. If the stimulus mode is patterns, the break would occur every fifth test. If, however, the stimulus mode is waveform (time-based) or tester emulation, the break interval could exceed five tests, since the circuit state may not stabilize by the end of each test.

The colon form must be used to cancel breakpoints based on stable states:

```
BREAK PSTEP:
```

The **TSTEP** keyword specifies a time interval. The command:

BREAK TSTEP=<*integer*>

causes periodic breaks from simulation with the specified interval (number of time-units). For example,

```
BREAK TSTEP=50
```

sets a breakpoint every fifty time-units.

The colon form must be used to cancel breakpoints based on elapsed time:

```
BREAK TSTEP:
```

Directing Break Messages

Whenever a break occurs, SIMIC issues one or more messages describing the cause or causes. By default, these messages are directed to the terminal. The command:

```
NO BREAK TERM:
```

disables break messages at the terminal. These messages can be re-enabled at the terminal with the command:

```
BREAK TERM:
```

Break messages can also be directed a file with the **FILE** keyword. The form:

BREAK FILE=<*file name*>

explicitly specifies a file name, and possibly its extension. The command:

```
BREAK FILE:
```

specifies a file with the default file name (see **DEFINE FILE**). If the second form is used, or if no file extension is specified in the first form, the file's default extension is **brk**.

Regardless of how they were enabled, break messages to a file are disabled with the command:

```
NO BREAK FILE:
```

Run Command: CLAMP (CL)

Function

The **CLAMP** command can be used to force signals to remain at specified values until explicitly released (with the **NO CLAMP** command), either for debugging or for circuit initialization.

The **CLAMP** command is used to define the functionality of the SIMIC PLA primitive (specifically, its AND plane and its OR planes), and may be used to define the functionality of BOOLEAN elements.

The **CLAMP** command is used to initialize the contents of the SIMIC ROM primitive, and may also be used to initialize the contents of the RAM primitives.

Usage

Forcing And Releasing Signal Values

Signals may be forced to logic-0, logic-1, X (unknown), or Z (floating unknown) with the respective commands:

CLAMP ZERO=<*signals*> TNUM=<*tnum*>
CLAMP ONE=<*signals*> TNUM=<*tnum*>
CLamp X=<*signals*> TNUM=<*tnum*>
CLAMP HIZ=<*signals*> TNUM=<*tnum*>

The specified signals will be clamped to the specified value at the start of the specified test. If the **TNUM** keyword option is omitted, the signals will be clamped to the specified value at the start of the next test, after simulation is resumed.

Clamped signals remain at their forced values until released with the **NO CLAMP** command. The **LIST** keyword, with either a list of signals or its colon form, may be additionally used to release signals regardless of forced value. For example:

```
NO CLAMP ONE=a,b TNUM=50
```

releases signals **a** and **b** from being forced to logic-1 at test 50 (but does nothing if they are not forced to this value). The command:

```
NO CLAMP ONE: TNUM=50
```

releases all signals forced to logic-1 at test 50, while:

```
NO CLAMP LIST: TNUM=50
```

releases all forced signals at test 50. If the **TNUM** keyword option is omitted, signals are released at the start of the next test, after simulation is resumed.

Defining PLA Functionality

A PLA's personality is defined by specifying the product terms of its AND plane and the sum terms of its OR plane.

The AND plane personality is set by the **CLAMP** command as follows:

```
CLAMP PART=<part_pla> $
    AND=<ninputs>*<nproducts> $
    BITMAP=<connection_map>
```

where ***<part_pla>*** is the PLA's part name, ***<ninputs>*** is the number of inputs to the AND plane (which must be the number of data inputs), and ***<nproducts>*** is the number of AND plane product terms (and the number of inputs to the OR plane). This keyword-field is used for error checking.

The ***<connection_map>*** contains ***<nproducts>*** items, each containing ***<ninputs>*** **0**, **1**, or **X** characters that define a single product term, ordered according to the PLA data inputs. These characters correspond to **complemented**, **true**, or **don't-care**, respectively.

The OR plane personality is set by the CLAMP command as follows:

```
CLAMP PART=<part_pla> $
    OR=<nproducts>*<noutputs> $
    BITMAP=<connection_map>
```

where ***<part_pla>*** is the PLA's part name, ***<nproducts>*** is the number of inputs to the OR plane (which must be the number of AND plane outputs), and ***<noutputs>*** is the number of OR plane sum terms (and the number of PLA outputs). This keyword-field is used for error checking.

The ***<connection_map>*** contains ***<noutputs>*** items, each containing ***<nproducts>*** **0**, **1**, or **X** entries that define a single sum term, ordered by the AND plane outputs. These characters correspond to **complemented**, **true**, or **don't-care**, respectively. The sum terms, in turn, are ordered by the PLA outputs.

For example, if a PLA with three data inputs, **a**, **b**, and **c** (plus **CS** and **EN**), two outputs, **sum** and **cout**, and instance name **fadd** implements a full-adder with equations:

$$\mathrm{sum} = a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$$

$$\mathrm{cout} = ab + bc + ac$$

then the following **CLAMP** commands personalize the PLA:

```
CLAMP PART=fadd AND=3*7 $
    BITMAP=100 010 001 111 11x x11 1x1
CLAMP PART=fadd OR=7*2 $
    BITMAP=1111xxx xxxx111
```

When the PLA's **EN** input is logic-0, the outputs are set to the ***disabled*** value specified by the **ENABLE** option of the **CLAMP** command:

```
CLAMP PART=<part_pla> ENABLE=<dis_val>
```

where ***<dis_val>*** can be either **1** or **0**. If unspecified, the value defaults to **0**. For example:

```
CLAMP PART=pla1 ENABLE=1
```

causes all outputs to be logic-1 whenever the enable, **`EN`**, is logic-0.

Defining BOOLEAN Functionality

The CLAMP command can be used to define the functionality of all instances of a BOOLEAN type or of a single BOOLEAN part with the respective commands:

```
CLAMP TYPE=<type_name> BOOLEAN=<boolean_block>
CLAMP PART=<part_name> BOOLEAN=<boolean_block>
```

where:

- ***<type_name>*** is the name of the BOOLEAN TYPE to modify.
- ***<boolean_block>*** is the ***BEGIN; equation; equation; ... END;*** format illustrated below.
- ***<part_name>*** is the name of a BOOLEAN instance to modify.

For example, if a BOOLEAN TYPE represents a full-adder having three inputs, **`a`**, **`b`**, and **`c`**, and two outputs, **`sum`** and **`cout`**, and the full-adder equations are:

$$sum = a\bar{b}\bar{c} + \bar{a}b\bar{c} + \bar{a}\bar{b}c + abc$$

$$cout = ab + bc + ac$$

then the following **`CLAMP`** commands would make every instance of this element have this functionality:

```
CLAMP TYPE=fadd BOOLEAN= $
     BEGIN ; $
     sum = a*^b*^c + ^a*b*^c + ^a*^b*c ; $
     cout = a*b + b*c + a*c ; end ;
```

If only instance **`fad5`** need be defined at run-time, the same command would be used, except the keyword field **`TYPE=fadd`** would be replaced with **`PART=fad5`**.

Initializing the Contents of a RAM

Normally, RAM contents are initialized during simulation by performing writes to the RAM. However, SIMIC also supports partial or complete initialization of the RAMA, RAMB, and RAMC primitives prior to simulation via the **`CLAMP`** command. The command structure is:

```
CLAMP PART=<part_ram> DATA=<addresses_data>
```

where ***<part_ram>*** is the RAM's part name, and ***<addresses_data>*** is the data to write into the RAM in the following format:

```
X<address> <data> X<address> <data> ...
```

where ***<address>*** is the starting address for the data that follows, in hexadecimal, and ***<data>*** is the data described in hexadecimal. For example:

```
CLAMP PART=RAM1 DATA= $
X0000     00 01 02 03 04 05 06 07 $
          08 09 0A 0B 0C 0D 0E 0F $
XC000     FF FF FF FF
```

Run Command: DEFINE (DE)

Function

The **DEFINE** command can be used to define global simulation parameters and the default file name.

The **DEFINE** command is used to define primary input stimuli.

The **DEFINE** command is used to define vector alias names that group signals and assign the group an output format for **PRINT/WRITE** operations.

The **DEFINE** command is used to initiate tester emulation mode and to define timing generators and strobes.

Usage

Defining Global Parameters

The **BTGDELAY** keyword controls whether loading is propagated through ON ideal switches (BTGN and BTGP) to dynamically adjust driver delays. By default, SIMIC performs dynamic delay modification. The command:

```
DEFINE BTGDELAY = STATIC
```

forces fixed delays at drivers of ideal switches. The command:

```
DEFINE BTGDELAY = DYNAMIC
```

enables dynamic delay modification.

The **FILE** keyword specifies the default file name:

DEFINE FILE = *<file name>*

The name, ***<file name>***, is prefixed to the default file extension associated with particular run commands to systematize file naming. It should *not* contain an extension. Example:

```
DEFINE FILE = mycircuit
```

The **NEAR** keyword defines the window for near hazard analysis. By default, this window is 2×(average gate propagation delay). The multiplier can be changed from its default value of 2 to another value with:

DEFINE NEAR = *<integer>*

The SIMIC default is equivalent to:

```
DEFINE NEAR = 2
```

The **OSCILLATION** keyword defines the threshold for excessive activity. By default, a signal's activity is excessive it executes 10 level transitions in response to a single change of input state. Excessively active signals are forced to X until the next input event. This definition can be changed with:

`DEFINE OSCILLATION =` *<integer>*

The SIMIC default is equivalent to:

`DEFINE OSCILLATION = 10`

The maximum specifiable value is 255. Oscillation checks can be disabled with the command:

`DEFINE OSCILLATION = INFINITE`

The **PULSE** keyword defines the width of pulse hazards. By default, this width is 3×(average gate propagation delay). The multiplier can be changed from its default value of 3 to another value with:

`DEFINE PULSE =` *<integer>*

The SIMIC default is equivalent to:

`DEFINE PULSE = 3`

The **RDEPTH** keyword defines the correspondence between the gate-level RESISTIVE drive strength and switch-level series depth. By default, this drive strength corresponds to a depth of 3. This default can be changed with:

`DEFINE RDEPTH =` *<integer>*

The SIMIC default is equivalent to:

`DEFINE RDEPTH = 3`

The **STABILITY** keyword controls whether decays are included in the definition of circuit state stability. The command:

`DEFINE STABILITY = POSTDECAY`

defines the circuit state to be stable only after all transitions to non-tristating values have occurred, and all tristated signals with finite decay have decayed, in response to a primary input event. This is the SIMIC default. The command:

`DEFINE STABILITY = PREDECAY`

defines the circuit state to be stable after all transitions to non-tristating values have occurred, in response to a primary input event.

The **XADDRESS** keyword controls handling of unknown (X) address lines at the ROM and RAM (RAMA, RAMB, and RAMC) SIMIC primitives. By default, if four or fewer address lines are unknown, SIMIC exhaustively reads all possibly addressed locations to set differing output lines to X, or writes data to all possibly addressed locations. If the number of unknown address lines exceeds this threshold, output data is all-X on a read, and the entire RAM contents are set to X on a write. The command:

`DEFINE XADDRESS =` *<integer>*

changes the threshold to the specified number. The SIMIC default is equivalent to:

`DEFINE XADDRESS = 4`

Defining Primary Input Stimuli

Primary input stimuli are defined using the command form:

```
DEFINE <pw><name>.<width><defaults>= $
                 <sequence>
```

where:

- ***<pw>*** is either a **P** (for patterns) or **W** (for waveforms),
- ***<name>*** is a user-defined name for the input sequence, immediately following the ***<pw>*** designator; combined, the two entries form the sequence's complete name,
- ***<width>*** is the number of signals for which the input sequence is defined,
- ***<defaults>*** (optional) specifies default attributes of the stimulus sequence appearing on the right side of the equal sign, and
- ***<sequence>*** is the stimulus sequence being defined.

The optional ***<defaults>*** on the left side of the equal sign are:

.<duration>.<format>.<strength>

where:

- ***<duration>***, called the **default duration**, is

  for patterns, the *number of tests* to maintain each input state of ***<sequence>*** before applying the next one. If unspecified, each input state is maintained for one test (the default duration is 1)

  for waveforms, the *amount of time* to maintain each input state of ***<sequence>*** before applying the next one. If unspecified, the default duration is 0, which means that all stimulus timing must be described in ***<sequence>***
- ***<format>*** is the default format (radix) of ***<sequence>*** (**BINARY**, **OCTAL**, **HEXADECIMAL**, **INTEGER**). If unspecified, the default is **BINARY**, and
- ***<strength>*** is the default strength of ***<sequence>*** (**POWER**, **DRIVING**, **RESISTIVE**, **FLOATING**). If unspecified, the default is **DRIVING**.

A single dot (.) separates adjacent options on the left side of the equal sign.

For example, the command:

```
define pall.3 = 000 001 010 011 100 101 110 111
```

defines a pattern, since the ***<pw>*** character is **P**. The pattern's name is, **pall**, and its width is **3**. Since there are no ***<duration>***, ***<format>***, or ***<strength>*** specifications following the width entry, the default duration of each stimulus state is one test, the radix of ***<sequence>*** is **BINARY**, and the stimuli have **DRIVING** strength. See Chapter 2.3 for a complete description of these options and specification of the sequence, ***<sequence>***.

Defining Vector Aliases

The **DEFINE** command may be used to group signals at run time, assign a name to each group, its **vector alias**, and associate a radix with each group for **PRINT/WRITE** output. The format for this command is:

DEFINE V*<name>*.*<format>*=*<signals>*

where ***<name>*** is a user-supplied alias for the vector, and ***<format>*** is one of the following format specifications:

- **LEVEL** – individual levels for each bit.
- **OCTAL** – octal representation.
- **HEXADECIMAL** – hexadecimal representation.
- **INTEGER1** (**INT1)** – One's complement representation.
- **INTEGER2** – Two's complement representation.
- **POSINTEGER** – Positive integer representation.

Except for the one's complement specification, which can only be **INTEGER1** or **INT1**, any valid specification prefix is sufficient.

For example, to display **a**, **b**, **c**, and **d** together in hexadecimal format, and **e**, **f**, **g**, and **h** together in integer format, the following commands can be used:

```
DEFINE VAB.HEX=a,b,c,d
DEFINE VEF.INT=e,f,g,h
PRINT LIST=vab*vef
```

Initiating Tester Emulation Mode

The **PERIOD** keyword defines a master period corresponding to a tester cycle. The start of each interval begins a new test. The default test period is defined with the command:

DEFINE PERIOD=*<time>*

where <***time***> is the number of time-units in each test period. Defining a period automatically switches SIMIC into tester emulation mode, and it will remain in tester emulation mode until the master test period is explicitly removed. This is accomplished by setting the value to 0 with the command:

```
DEFINE PERIOD=0
```

Defining Timing Generators

This section presents a syntactic overview of defining timing generators. See Chapter 2.8 for a complete description.

Timing generator formats and drive envelopes (time-sets) are specified with the **DEFINE** command. If the primary input does not tristate, the following format may be used to define a timing generator:

DEFINE T*<name>*.*<dformat>*=*<dmarks>*

where ***<name>*** is a user-defined name for the time-set, ***<dformat>*** is one of the drive mask types listed below, and ***<dmarks>*** is a list of marks, or timing values, whose order and number depends on the ***<dformat>*** selected:

1. **NRZ** (Non-Return-to-Zero).
2. **RZ** (Return-to-Zero)
3. **RO** (Return-to-One).
4. **RC** (Return-to-Complement)

For example,

```
DEFINE TDAT1.NRZ=20,40
```

defines a timing generator named **tdat1** with a NRZ format that delays transition to logic-1 by 20 time-units (from the beginning of the tester cycle), and transitions to logic-0 by 40 time-units.

The correct number of timing marks on the right, and their significance, depends on the selected drive mask.

If the primary input does tristate, the following format may be used to define a timing generator:

DEFINE T*<name>*.*<dformat>*.*<eformat>*= $
<dmarks>;*<emarks>*

where the additional <***eformat***> option is one of the enable masks listed below, controlling the mode of driving/ tristating transitions, and the additional <***emarks***> field specifies the times at which these transitions should occur:

1. **NE** (No-Envelope)
2. **RD** (Return-to-Drive)
3. **RF** (Return-to-Float)

For example,

Note the addition of ***<eformat>***, which selects one of the enable formats described above, and ***<emarks>*** which is a list of timing values, whose order and number depends on the selected enable format.

```
DEFINE TDAT2.NRZ.NE=20,40;50,70
```

defines a NRZ timing generator named **tdat2** with the same characteristics as **tdat1** above for driven transitions. However, for **tdat2**, transitions from driving to floating are delayed 50 time-units (from the beginning of the tester cycle), and transitions from floating to driving are delayed 70 time-units.

Defining Timing Generators

SIMIC supports two different types of strobes. The first is a point (or edge) strobe (**SP**), and the second is a window strobe (**SW**). The format for defining a strobe is:

```
DEFINE S<name>.<sformat>=<smarks>
```

where ***<name>*** is the user-defined name for the strobe, ***<sformat>*** is either **SP** (for a point strobe), or **SW** (for a window strobe), and ***<smarks>*** is a single value for **SP**, indicating the time to fire the strobe, or two values for **SW**, indicating the time to start and the time to stop the window strobe, respectively. All times are in time-units, relative to the start of the period.

For example, the following command defines a window strobe, whose window begins at time 600 and ends at time 700:

```
DEFINE SDEMO1.SW=600,700
```

To define a point strobe at time 800:

```
DEFINE SDEMO2.SP=800
```

Run Command: EXECUTE (EX)

Function

The **EXECUTE** command is used to execute run command files.

Usage

The **FILE** keyword specifies the name of the run command file to execute. The command form:

```
EXECUTE FILE:
```

causes the run commands in the file having the default file name and default extension **run** to be executed. The command form:

EXECUTE FILE=*<file list>*

Here, ***<file list>*** explicitly specifies one or more run command files to be executed.

Executed run files may themselves contain **EXECUTE** commands. The maximum level of **EXECUTE** command nesting is 4.

Run Command: GET (GE)

Function

The **GET** command is used to compile textual network descriptions and load the resulting binary representation.

The **GET** command is used to load a previously-compiled representation.

The **GET** command is used to read backannotation data and recompute propagation delays to reflect net loading.

The **GET** command is used to load minimum, typical, or maximum delay sets.

Usage

GET command options are explained in Chapter 2.2.

Circuit Compilation Options

The **TYPE** keyword specifies the **main type**, the top-level circuit to be simulated. If only the default network description file need be read (default file name, file extension of **net**), the circuit could be compiled with:

GET TYPE=<*main type*>

The **FILE** keyword may be used to specify the file(s) containing the network description:

GET TYPE=<*main type*> $
    FILE=<*file name1*>,<*file name2*> ...

Any number of network description files may be specified; the files are read in the order they appear in the list. For example:

```
GET TYPE=full-adder FILE=cellib,mycircuit
```

causes files **cellib** and **mycircuit** to be read, in that order.

The **TYPE** keyword initiates compilation. The **FILE** keyword option, and the keyword options described below, must be specified either prior to the **GET TYPE** command, or in the same command.

By default, SIMIC aborts compilation if 20 circuit description errors have been found. This default can be changed with the **STOP** keyword:

GET STOP=<*integer*>

The SIMIC default is equivalent to:

```
GET STOP=20
```

The **`LFILE`** keyword specifies the name of the **listing file** to be generated after compilation. By default, no listing file is generated. The command:

`GET LFILE:`

specifies that the listing file's name should have the default file name and the default listing file extension, **lst**. The command:

`GET LFILE=`<*file name*>

explicitly specifies the listing file's name.

The **`REPORT`** keyword controls the amount of information to be written to the listing file. The command:

`GET REPORT= SYMBOLS`

specifies that only part and signal names be written to the listing file. Any prefix of the word SYMBOLS is valid. The command:

`GET REPORT= ALL`

specifies that in addition to these names, topological and electrical information be written to this file as well.

The request to generate a listing file can be cancelled with either:

`NO GET LFILE:` or `NO GET REPORT:`

By default, SIMIC does not create a binary file containing the compiled circuit description. The SFILE keyword specifies that this file be generated. The command:

`GET SFILE:`

assigns this file the default file name and the default file extension, **rnt**. The command:

`GET SFILE=`<*file name*>

explicitly names this file.

Once this file is created, the circuit description can be quickly retrieved in subsequent simulation sessions with the **`RFILE`** keyword option, without requiring re-compilation of the network description.

A previous **`GET SFILE`** command can be cancelled with:

`NO GET SFILE:`

Loading A Previously-Compiled Circuit Description

The **`RFILE`** keyword option restores a previously-compiled description. The command:

`GET RFILE:`

restores the circuit description contained in the file whose name is the default file name and whose extension is **rnt**. The command:

`GET RFILE=`<*file name*>

explicitly names this file.

Updating Wiring Delays With Backannotation Data

The **`AFILE`** keyword option causes a backannotation file (containing net **`LOAD`** elements) to be read, and driver delays to be updated accordingly. The command:

`GET AFILE:`

causes the file with default file name and default extension **ann** to be read, while the command:

`GET AFILE=`*<file name>*

explicitly names this file.

Updating Wiring Delays With Backannotation Data

Either of three tables (delay sets), corresponding to **`TYPICAL`**, **`MINIMUM`**, and **`MAXIMUM`** timing values may be loaded with the **`TIMING`** keyword option. The command is:

`GET TIMING=`*<table>*

where ***<table>*** is either **`TYPICAL`**, **`MINIMUM`**, or **`MAXIMUM`**.

Run Command: HISTORY (HI)

Function

The **HISTORY** command is used to save simulation event history to interface with post-simulation display and analysis programs. Two history files are created; the general history file and the sequential history file. Their default extensions are **hig** and **his** respectively. By default, history files are not created.

Usage

The **HISTORY** command is described in Chapter 2.9.

The **HISTORY** command can be restricted to a specified interval of tests, for patterns, or time, for waveforms, with the **PRANGE** keyword option:

`HISTORY PRANGE=`*<prange spec>*

Specifying File Name

The **FILE** keyword option is used to specify the filename components of the two history files. The command:

`HISTORY FILE:`

assigns the default file, while the command:

`HISTORY FILE=`*<file name>*

assigns a file name explicitly. Note: *no* file extension should be specified in this form of the command.

Specifying Signals To Trace

The **LIST** keyword selects the signals for which events are to be saved. History file creation is enabled by this selection. The signals specified with this keyword are subject to name-based filtering (see STRING keyword option below); only unfiltered signals are accepted for history trace. The command:

`HISTORY LIST:`

selects all unfiltered signals, while the command:

`HISTORY LIST=`*<signals>*

selects all signals in the specified list that are unfiltered.

The **NO** prefix can be used to selectively remove signals. Signals specified with the **NO** prefix are not filtered. For example, the sequence of commands:

```
HISTORY LIST:
NO HISTORY LIST=abc
```

causes all unfiltered signals to be traced except signal **abc**.

Specifying A Dump Interval

Once enabled, history dumps (snapshots) are performed every 100 test steps (by default). This default can be changed with the command:

HISTORY PSTEP=*<integer>*

where ***<integer>*** is the test interval to perform the dumps. History dumps can be disabled with the command:

NO HISTORY PSTEP:

Name-Based Filtering

The **STRING** keyword may be used to systematically filter signals from subsequent **HISTORY LIST** specifications.

Filtering based on the name of the part generating each signal can be specified with one of two complementary options. The command:

NO HISTORY STRING=&NUMERIC

will cause any signals generated by a part whose lowest-level name component is purely numeric (e.g., **a.b.c.153**) to be filtered. Alternatively, the command:

NO HISTORY STRING=&ALPHANUMERIC

will cause any signals generated by a part whose lowest-level name component is not a pure numeric (e.g., **a.b.c.d12**) to be filtered.

Filtering based on the signal's hierarchical name can be specified with the command:

NO HISTORY STRING=*<filter string>*

where ***<filter string>*** is a pattern of characters, possibly enclosed in quotes. For example, the command:

NO HISTORY STRING="#"

will filter all signals in subsequent **HISTORY LIST** specifications whose names contain the character "#".

The general form of signal-name based filtering is:

NO HISTORY STRING= *<integer>***<filter string>*

where ***<integer>*** specifies a filter string repetition factor. For example:

NO HISTORY STRING=2*"."

causes all signals at a hierarchical level of three or higher (their names containing at least 2 dots) to be filtered from subsequent **HISTORY LIST** specifications.

Using the **STRING** keyword without the **NO** prefix removes a filter. The command:

HISTORY STRING="#"

causes subsequently specified signals containing "#" in their names to be traced.

Run Command: LOOK (LO)

Function

The **LOOK** command is used to interactively probe signal values while the state of the simulated circuit is frozen in time.

Optionally, information may also be requested about the inputs to the element generating the signal (if the signal is not a primary input) and/or the loads driven by the signal.

Usage

The **LOOK** command is described in the Section *Probing For Signal State Information* in Chapter 2.6.

Probing Specific Signals

The LIST keyword is used to specify the signals of interest: The command:

`LOOK LIST=`*<signals>*

causes the current values of the specified signals to be reported at the terminal. The **LIST** keyword option is *not* sticky; only the values of currently specified signals are reported. For example:

```
>>: LOOK LIST=A
At Time= 23, Test=101:
A= '0' [Primary Input]
>>: LOOK LIST=CARRY-OUT
At Time= 23, Test=101:
CARRY-OUT= '1' [OR]
```

The command:

```
LOOK LIST:
```

produces a table of all signals states (in alphanumeric sequence). The table's format is equivalent to the **PRINT** (or **WRITE**) "dump" format, discussed in the Subsection *Selecting Signals to Output* in Chapter 2.4.

For internal signals, the **INPUTS** keyword option causes additional information to be reported on the inputs of the elements that generate the signals. The command form is:

```
LOOK INPUTS:
```

For example:

```
>>: LOOK INPUTS: LIST=CARRY-OUT
AT TIME= 23, TEST= 101:
CARRY-OUT= '1' [OR]
I:= AND1= '0' [AND]
I:= AND2= '1' [AND]
I:= AND3= '0' [AND]
```

In this example, the **carry-out** signal is a logic-1, and it is generated by an OR gate. This gate has three inputs, **and1**, **and2**, **and3**, each the output of an AND gate. The second input, **and2**, is causing the logic-1 at **carry-out**.

The **INPUTS** keyword option is sticky; once enabled, each subsequent **LOOK LIST** command will also contain element input information for internal signals.The command:

```
NO LOOK INPUTS:
```

removes this information from subsequent **LOOK** output.

Fanout information can also be included in the **LOOK LIST** output. with the command:

```
LOOK OUTPUTS:
```

For example:

```
>>: LOOK OUTPUTS: LIST=AND2
At Time= 23, Test= 101:
AND2= '1' [AND]
O:= CARRY-OUT [OR]
```

additionally reports that **and2** fans out to the OR gate named **carry-out**.

The **OUTPUTS** keyword option is sticky; once enabled, each subsequent **LOOK LIST** command will also contain the instance names of all parts driven by the traced signal. Fanout information can be subsequently removed from **LOOK** output with the command:

```
NO LOOK OUTPUTS:
```

Displaying All Signals At A Specified State

Many times it is useful to see if there are any signals at an 'X' or 'Z' state. To display all signals at an 'X' state, use the command:

```
LOOK X:
```

Similarly, to display all signals at a 'Z' (tristate) value, issue the command:

```
LOOK HIZ:
```

The format of both reports is identical to the **LOOK LIST=<*signals*>** command's report format. These two commands do not honor the **INPUTS:** or **OUTPUTS:** keyword options, if in effect.

Run Command: PRINT (PR)

Function

The **PRINT** command is used to report selected signal values in a tabular format during simulation. Output is directed to the terminal.

Usage

The **PRINT** command is described in Chapter 2.4.

The **PRINT** command can be restricted to a specified interval of tests, for patterns, or time, for waveforms, with the **PRANGE** keyword option:

PRINT PRANGE=<*prange spec*>

Specifying Signals and Format

The **LIST** keyword is used to select signals and format the output: The command form for this selection is:

PRINT LIST=<*signals and format*>

where ***<signals and format>*** has the form of a ***<signals>*** specification, possibly augmented with formatting options. These options are:

1. Inserting one or more blank vertical columns between signals. This is accomplished by entering an asterisk (*****) for each blank column.
2. Forcing a new row in the table. This is accomplished by entering a pound sign (**#**) at the desired point.

Commas or whitespace are optional before or after ***** or **#**.

For example:

```
PRINT LIST=u,v,*w**x#y
```

will cause the simulation output to consist of the value of signal **u**, followed by the value of **v**, followed by a blank column, followed by the value of **w**, followed by two blank columns, followed by the value of **x**. The value of signal **y** will be output at the first position of the next line.

The second form of the **LIST** keyword:

```
PRINT LIST:
```

specifies that the values of *all* signals be reported in "dump" format; groups of five values ordered alphabetically, according to the **symbol** section of the listing file.

The **PRINT** command's **LIST** keyword is "sticky"; any signal specified will continue to be reported until explicitly removed. Removing signals from the list is accomplished with the **NO** prefix. For example:

```
NO PRINT LIST=a,b
```

removes signals **a** and **b** from the list of reported signals. The format options "*" and "#" are not specified with the **NO** prefix. The command:

```
NO PRINT LIST:
```

removes all signals, thereby terminating **PRINT** output.

Suppressing Signal Strength

By default, a 15-character set is used to represent combined level and strength information. Optionally, output can be restricted to a 4-character set representing levels; **0**, **1**, **X**, and **Z** (although **Z** represents the combination of unknown level at floating strength). The **VALUES** keyword option is used to select value representation. The command:

```
PRINT VALUES=LEVELS
```

selects the 4-character representation, while the command:

```
PRINT VALUES=STRENGTHS
```

restores the 15-character default representation.

Requesting Output At Stable Points

The default **PRINT** operation is to output the requested simulation values each time the circuit becomes stable. The **PSTEP** (**PS**) keyword option can be used to control frequency of this output, or inhibit it. The command:

PRINT PSTEP=*<integer>*

changes this interval. For example:

```
PRINT PSTEP=5
```

specifies that output occur every fifth stable point.

Output based on attaining a stable state can be disabled with the **NO** command prefix:

```
NO PRINT PSTEP:
```

Requesting Time-Periodic Output

Output can be requested at specified time intervals using the **TSTEP** keyword. The command:

PRINT TSTEP=*<integer>*

will cause an output to occur every ***<integer>*** simulation time-units. If the time-step specification is preceded by a plus ("+") sign, output will continue regardless of circuit stability. For example:

```
PRINT TSTEP=+100
```

produces output every 100 time-units, even when the circuit state is stable.

The **NO** prefix can be used to disable time-periodic output:

```
NO PRINT TSTEP:
```

The **BEGIN** keyword option may be used to skew (offset) the first **TSTEP** output. This command has the form:

```
PRINT BEGIN=<integer>
```

where ***<integer>*** specifies the time offset to the first output.

To remove the offset, specify a value of 0:

```
PRINT BEGIN=0
```

Requesting Output Based On Activity

The **CHANGE** keyword option specifies that output be triggered by activity at monitored signals. The command form is:

```
PRINT CHANGE=<signals>
```

where ***<signals>*** is a subset of the signals currently being monitored with the **PRINT** command (other signals are ignored). This command specifies that output should occur whenever any of the specified signals changes state.

The command:

```
PRINT CHANGE:
```

will trigger print output when any of the monitored signals changes state.

Subsequently, the command:

```
NO PRINT CHANGE=<signals>
```

would inhibit the specified signals in ***<signals>*** from functioning as triggers, while:

```
NO PRINT CHANGE:
```

would disable all activity-based output.

Header And Tabular Format Control

SIMIC, by default, limits each output line to 80 columns. The **EXPAND** keyword option can be used to modify output line width. The command form:

```
PRINT EXPAND:
```

expands output line width to 132 columns, while the command form:

```
PRINT EXPAND=INFINITE
```

allows arbitrary output line width.

The **HEADER** keyword option controls whether the simulation header, consisting of the simulation options and signal names, be output prior to the tabular simulation values. By default, the header is enabled. The command:

```
NO PRINT HEADER:
```

disables the header output, and the command:

```
print header:
```

enables it.

When waveform stimuli are used, the test number field of the simulation output can be optionally suppressed with the **TNUM** keyword option. By default, this field is present in every record. The command:

```
NO PRINT TNUM:
```

suppresses the test field. Each output line will contain the time, followed by a colon, followed by the requested signal values.

The test field can be restored with the command:

```
PRINT TNUM:
```

Run Command: ? (query simulation parameters)

Function

The **?** command is used to obtain information on circuit parameters and on current simulation options.

Usage

Each query consists of a question mark (?) followed by a parameter keyword specifying the type of request. Optional whitespace may delimit the question mark and keyword. Requests for circuit parameter values also require the **LIST** or **PARTS** keyword, whichever is appropriate, to identify the signals or elements of interest.

The **CHECK** keyword requests the current values of timing check parameters, and whether a **WARN**, **BREAK**, or **XPROPAGATE** command option is currently active for any primitive that supports these checks. The command:

    ?CHECK PARTS=<parts_ff>

requests this information for the parts specified in ***<parts_ff>***, while:

    ?CHECK PARTS:

requests this information for all parts supporting timing checks.

The **DECAY** keyword can be used to request the current decay time of any signal in the circuit. The command:

    ?DECAY LIST=<signals>

requests decay times of the specified signals, while:

    ?DECAY LIST:

requests the decay time of every signal.

The **DEFINE** keyword can be used to request the current values of global simulation parameters and definitions. The command form is:

    ?DEFINE

The **DELAY** keyword can be used to request the current rise and fall delays of any signal in the circuit. The command:

    ?DELAY LIST=<signals>

requests the delays of the specified signals, while:

    ?DELAY LIST:

requests the delay of every signal.

The **LOADING** keyword requests signal loading information. The command:

    ?LOADING LIST=<signals>

requests the loading at the specified signals, while:

```
?LOADING LIST:
```

requests the loading at every signal.

The **PRINT** keyword can be used to determine the signals currently being reported by the **PRINT** command. The command form is:

```
?PRINT
```

The **SPIKE** keyword can be used to request the current values of the spike generation parameters (**FILTER** and **LIBERAL** attributes) of any signal in the circuit. The command:

```
?SPIKE LIST=<signals>
```

requests the spike control parameter values of the specified signals, while:

```
?SPIKE LIST:
```

requests the spike control parameter values of every signal.

The **WRITE** keyword can be used to determine the signals currently being reported by the **WRITE** command. The command form is:

```
?WRITE
```

Run Command: QUIT (QU)

Function

The **`QUIT`** command is used to exit SIMIC.

Usage

A SIMIC session is terminated with the command:

```
QUIT
```

Run Command: RESTORE (RE)

Function

The **`RESTORE`** command is used to restore a specified circuit state to be used as the initial state for subsequent simulation.

Usage

The **`TNUM`** keyword is used to specify the state to be restored. If the circuit state is restored from a checkpoint file, this file is assumed to have the default file name and the default extension **sav**. The **`FILE`** keyword may be used to explicitly specify the checkpoint file's name.

Restoring The Initial Unknown State

The command:

`RESTORE TNUM=0`

restores the circuit to its initial state prior to simulation (with all signal values unknown), the simulation time to 0, and the Test number to 1.

Restoring The Last Stable State

The command:

`RESTORE TNUM=*`

restores the circuit to its most recent stable state, and the simulation time and Test number to their values at that time.

Restoring A Checkpoint State

The command:

`RESTORE FILE=`*<file name>*

may be used to specify the checkpoint file if this is not the default file.

The command:

`RESTORE TNUM=?`

may be used to determine the test numbers corresponding to all saved states in the checkpoint file.

The command:

`RESTORE TNUM=`*<tnum>*

restores the state previously saved in the checkpoint file at Test ***<tnum>***.

By default, the Test number and simulation time are also restored to their values at the time the state was saved. The command:

```
NO RESTORE PRANGE:
```

overrides this default, and forces the Test number and simulation time to remain at their current values after the saved state is restored. For example, the command sequence

```
NO RESTORE PRANGE:
RESTORE TNUM=500
```

causes the state previously saved at Test 500 to be restored, but does not affect the current Test number and simulation time.

See the Section *Replaying Portions of the Simulation* in Chapter 2.6 for more information on the **RESTORE** command.

Run Command: SAVE (SA)

Function

The **SAVE** command is used to create a checkpoint file. Only stable circuit states are saved. Saved states can be reloaded with the **RESTORE** command.

Usage

The **SAVE** command can be restricted to a specified interval of tests, for patterns, or time, for waveforms, with the **PRANGE** keyword option:

SAVE PRANGE=<*prange spec*>

By default, the checkpoint file has the default file name and the default extension **sav**. This file can be explicitly named with the **FILE** keyword:

SAVE FILE=<*file name*>

The **PSTEP** keyword option initiates saving circuit states. This command specifies an interval (count) of stable states over which saves should occur:

SAVE PSTEP=<*integer*>

For example:

```
SAVE PSTEP=250
```

causes every 250-th stable state to be saved in the checkpoint file. For pattern stimuli, this corresponds to saving the circuit state every 250 tests. This interval may be greater for waveform and timing generator stimulus modes, since the circuit state may not stabilize by the end of every stimulus state.

The special zero value:

```
SAVE PSTEP=0
```

causes the current state to be saved in the checkpoint file, if it is stable.

The command:

```
NO SAVE PSTEP:
```

disables saving states in the checkpoint file.

See the Section *Replaying Portions of the Simulation* in Chapter 2.6 for more information on the **SAVE** command.

Run Command: SET (SE)

Function

The **SET** command can be used to force signals to remain at specified values for a single test.

The **SET** command can be used to modify the delays of specified signals at run time.

The **SET** command can be used to modify decays at specified signals at run time.

The **SET** command can be used to modify timing check parameters at supported elements.

Usage

Forcing Signal Values

Signals may be forced to logic-0, logic-1, X (unknown), or Z (floating unknown) with the respective commands:

```
SET ZERO=<signals> TNUM=<tnum>
SET ONE=<signals> TNUM=<tnum>
SEt X=<signals> TNUM=<tnum>
SET HIZ=<signals> TNUM=<tnum>
```

The specified signals will be forced to the specified value at the start of the specified test. If the **TNUM** keyword option is omitted, the signals will be forced to the specified value at the start of the next test, after simulation is resumed. At the end of the test, the forced signals are released to assume consistent states.

Values to be **SET** at specified signals are scheduled in a queue. The **NO** prefix may be used to cancel specific **SET** operations. The **LIST** keyword, with either a list of signals or its colon form, may be additionally used to release signals regardless of forced value. For example:

```
NO SET ONE=a,b TNUM=50
```

cancels a previous **SET** command forcing signals **a** and **b** to logic-1 at test 50 (but does nothing if they are not forced to this value). The command:

```
NO SET ONE: TNUM=50
```

cancels all **SET**s forcing signals to logic-1 at test 50, while:

```
NO SET LIST: TNUM=50
```

cancels all **SET** commands that force any values at test 50. If the **TNUM** keyword option is omitted in the **NO SET** command, the next test is implied.

Modifying Signal Delays

The **FALL** keyword option allows modification of the specified signal's fall delays, without affecting their rise delays. The **RISE** keyword option allows modification of the specified signals' rise delays, without affecting their fall delays. The **CHANGE** keyword option sets both the rise and fall delays of the specified signals to the specified value:

SET FALL=*<delay spec>* LIST=*<signals>*

SET RISE=*<delay spec>* LIST=*<signals>*

SET CHANGE=*<delay spec>* LIST=*<signals>*

The **LIST** keyword's colon form can also be used to specify "all signals".

The delay specification ***<delay spec>*** can either be an integer or an integer percentage; an integer between 0 and 100, inclusive, followed by a percent sign (%), and optionally preceded by a plus (+) or minus(-) sign.

An integer specifies an absolute value. For example,

```
SET RISE=20 FALL=15 LIST=a,b
```

sets the rise delays of signals **a** and **b** to 20 and their fall delays to 15, while:

```
SET CHANGE=1 LIST:
```

sets all rise and fall delays to one time-unit.

An integer percentage specifies a change relative to the current delay value. For example:

`SET CHANGE=10% LIST:` or `SET CHANGE=+10% LIST:`

increases all rise and fall delays by 10%, while:

```
SET RISE=-20% FALL=-15% LIST=a,b
```

decreases the rise delays of signals **a** and **b** by 20%, and their fall delays by 15%.

If a signal has been specified that has multiple drivers (wire-tie), then all drivers will be modified accordingly.

Modifying Signal Delays

The **DECAY** keyword option allows modification of the specified signal's decay times. The command form is:

SET DECAY=*<decay spec>* LIST=*<signals>*

The **LIST** keyword's colon form can also be used to specify "all signals".

The decay specification, ***<decay spec>***, can be an integer or an integer percentage; an integer between 0 and 100, inclusive, followed by a percent sign (%), and optionally preceded by a plus (+) or minus(-) sign. Additionally, this decay specification can be any valid prefix of the word **INFINITE**:

SET DECAY=INFINITE LIST=*<signals>*

An integer specifies an absolute value. For example,

```
SET DECAY=500 LIST=a,b
```

sets the decay time at signals **a** and **b** to 500.

An integer percentage specifies a change relative to the current delay value. For example:

`SET DECAY=10% LIST:` or `SET DECAY=+10% LIST:`

increases the decay at all signals by 10%, while

```
SET DECAY=-10% LIST:
```

decreases the decay at all signals by 10%.

The **INFINITE** value sets the specified signals' decay times to infinite: For example:

```
SET DECAY=INFINITE LIST:
```

sets all decays to infinite.

Modifying Functional Timing Check Parameters

The SET command can be used to modify timing-check parameter values. The run command form:

SET PARTS: *<timing-check>*=*<delay spec>*

modifies the specified timing check parameter values in all parts, and

SET PARTS=*<parts_ff>* *<timing-check>*=*<delay spec>*

does so for the selected parts.

The keyword, ***<timing-check>***, is the designated timing-check name (e.g., **SETUP**, **HOLD**, etc.), and ***<delay spec>*** is the specified value, which can either be an integer or an integer percentage; an integer between 0 and 100, inclusive, followed by a percent sign (%), and optionally preceded by a plus (+) or minus(-) sign.

An integer specifies an absolute value. For example:

```
SET PART=F1 PW.C.L=10
```

assigns the value 10 time-units to the pulsewidth check parameter for low clock.

An integer percentage specifies a change relative to the current delay value. For example:

```
SET PART: SETUP=+30% HOLD=-20%
```

increases all setup timing checks by 30% and decrease all hold timing checks by 20%.

Run Command: SIMULATE (SI)

Function

The **SIMULATE** command initiates simulation.

Usage

The **SIMULATE** command is issued after all simulation options have been specified. The command is:

```
SIMULATE
```

The **SIMULATE** command can be restricted to a specified interval of tests, for patterns, or time, for waveforms, with the **PRANGE** keyword option:

SIMULATE PRANGE=<*prange spec*>

Run Command: TGEN (TG)

Function

The **TGEN** command causes the **tester interface file** to be created.

Usage

The **TGEN** command is described in the Section *Test Program Output* in Chapter 2.8.

By default, the tester interface file is not created. The **FILE** keyword option specifies a name for this file, thereby causing it to be generated during simulation. The command form:

    TGEN FILE:

specifies that the file's name be the default file name and the file's extension be the default extension **tgn**. The command form:

TGEN FILE=*<file name>*

explicitly specifies the file's name.

The **DISCONNECT** keyword option controls the signal values written to the tester interface file for bidirectional primary busses. By default, SIMIC writes either (a) the value that the bus would have if the primary drive were disconnected, when there is no wire-tie conflict, or (b) the actual value of the bus when there is a conflict. The command:

    NO TGEN DISCONNECT:

causes the actual value of the primary bus to be written (always including the effect of the primary drive), while the command:.

    TGEN DISCONNECT=ALL

causes the written value to always be the bus value that would result if the primary drive were disconnected. If the default is overridden, it can be subsequently restored with the command:

    TGEN DISCONNECT:

The **TARGET** keyword option specifies the target tester, which is written to the tester interface file. The command form is:

TGEN TARGET=*<name>*

where ***<name>*** is the tester's name. If unspecified, then the name defaults to "???".where ***<name>*** is the tester's name.

The **HIZ** keyword option defines a threshold depth such that any value at this depth, or at a weaker depth, be reported as "Z" in the tester interface file. The command form is:

TGEN HIZ=*<integer>*

where the integer value is less than 32,768. For example:

```
TGEN HIZ=20000
```

causes any value whose depth is 20,000 or higher to be written as “Z”.

Run Command: TRACE (TR)

Function

The **TRACE** command is used to trace circuit activity and indicate possible event causality.

Usage

The **TRACE** command is described in the Section *Tracing Circuit Activity* in Chapter 2.6. By default, signals are not traced.

The **TRACE** command can be restricted to a specified interval of tests, for patterns, or time, for waveforms, with the **PRANGE** keyword option:

`TRACE PRANGE=`*<prange spec>*

The **LIST** keyword option selects signals to be traced, thereby enabling trace reports. The command form:

`TRACE LIST=`*<signals>*

selects individual signals for tracing, while the command form:

`TRACE LIST:`

specifies that all signals be traced.

The **NO** command prefix is used to disable signal event tracing. The command:

`NO TRACE LIST=`*<signals>*

selectively disables event tracing for the specified signals, while:

`NO TRACE LIST:`

disables all event tracing.

The **EXPAND** keyword option specifies that causality information be additionally reported for each traced event. The command:

`TRACE EXPAND:`

globally enables causality information in subsequent trace reports, while the command:

`NO TRACE EXPAND:`

disables causality information. By default, trace output does not contain causality.

The **BEGIN** keyword option inhibits trace output from the beginning of a test to the specified time. When used with pattern stimuli, this allows “slow” sections of the logic to be located quickly. The command:

`TRACE LIST: BEGIN=`*<time>*

causes events occurring earlier the specified time within each test to be filtered from the trace output. The command:

```
NO TRACE BEGIN:
```
disables the **BEGIN** keyword option.

By default, **TRACE** messages are displayed at the terminal. This output is controlled with the **TERM** keyword. The command:
```
NO TRACE TERM:
```
disables trace output to the terminal, while the command:
```
TRACE TERM:
```
re-enables trace output to the terminal.

Additionally, the **FILE** keyword option can be used to independently direct trace output to a file. The command form:
```
TRACE FILE:
```
designates the file with default file name and default extension **trc**. The command form:

`TRACE FILE=`<*file name*>

explicitly names this file. The **NO** prefix can be used to disable trace output to a file:
```
NO TRACE FILE:
```

Run Command: WARN (WA)

Function

The **WARN** command is used to monitor the circuit during simulation and obtain messages concerning the occurrence of undesirable situations such as timing problems, switching hazards, wire-tie conflicts, and oscillations.

Previously-set **WARN** conditions are cancelled using the **NO** prefix. Unless explicitly indicated, keyword options that support the ***keyword=value*** field format also support this format with the **NO** prefix. For example:

```
NO WARN SPIKE=sig1
```

cancels the specific request to generate a warning message whenever a spike hazard occurs at signal **sig1**. All keyword options support the ***keyword:*** field format with the **NO** prefix. For example:

```
NO WARN SPIKE:
```

cancels all warning messages for the occurrence of spike hazards.

Warning messages for oscillations (**OSCILLATION**), wire-tie conflicts (**CONFLICT**), strobe errors (**STROBE**), primary input changes while the circuit is unstable (**UNSTABLE**), and all part timing checks (**SETUP**, **HOLD**, **PW**) are enabled by default. All other warnings are initially disabled.

Usage

See the Sections *Setting Simulation Warnings* and *Setting Simulation Breakpoints* in Chapter 2.6 for a complete description of the **WARN** command.

The **WARN** command can be restricted to a specified interval of tests, for patterns, or time, for waveforms, with the **PRANGE** keyword option:

WARN PRANGE=<*prange spec*>

Warn On A Transition To Unknown

The **MEMLATCH** keyword specifies a transition from a known state to an unknown state at a SIMIC primitive D-latch or edge-triggered flip-flop due to a sensitized unknown input (e.g., unknown clock):

WARN MEMLATCH=<*signals_ff*>

triggers a warning when an unknown input value is sensitized at a built-in memory element whose output is one of the specified signals, while:

```
WARN MEMLATCH:
```

triggers a warning when the state of any built-in memory element becomes unknown due to a sensitized unknown input.

The **X** keyword specifies a level transition from a known state to an unknown state, 1→X, 0→X:

`WARN X=`*<signals>*

triggers a warning when any of the specified signals becomes unknown, while:

`WARN X:`

triggers a warning when any signal becomes unknown.

Warn On The Occurrence Of Timing Hazards

The timing hazards described in this section are primarily due to the distribution of propagation delays within the circuit.

The **HAZARD** keyword is a shorthand designation of all switching hazards—**pulse**, **spike**, or **near**. The command:

`WARN HAZARD=`*<signals>*

triggers a warning when a pulse, spike, or near hazard occurs at any of the specified signals. For example, the command:

`WARN HAZARD=sig1`

is equivalent to:

`WARN PULSE=sig1 SPIKE=sig1 NEAR=sig1`

The command:

`WARN HAZARD:`

triggers a warning when a pulse, spike, or near hazard occurs at any signal.

The **MEMSPIKE** keyword specifies the occurrence of a **spike hazard** at a flip-flop primitive, and is therefore a subset of **SPIKE**. It narrows focus to spikes that could cause steady-state errors. The command:

`WARN MEMSPIKE=`*<signals_ff>*

triggers a warning when a spike hazard occurs at any of the specified flip-flop output signals, while:

`WARN MEMSPIKE:`

triggers a warning when a spike hazard occurs at any flip-flop. When a spike occurs at a flip-flop for which the **MEMSPIKE** keyword option has been specified, the warn message will contain *memspike* as the cause of the warning, rather than *spike*.

The **NEAR** keyword specifies the occurrence of a **near hazard** (see also **DEFINE NEAR**). The command:

`WARN NEAR=`*<signals>*

triggers a warning when a near hazard occurs at any of the specified signals, while:

`WARN NEAR:`

triggers a warning when a near hazard occurs at any signal.

The **PULSE** keyword specifies the occurrence of a **pulse hazard** (see also **DEFINE PULSE**). The command:

WARN PULSE=<*signals*>

triggers a warning when a pulse hazard occurs at any of the specified signals, while:

WARN PULSE:

triggers a warning when a pulse hazard occurs at any signal.

The **SPIKE** keyword specifies the occurrence of a **spike hazard**. The command:

WARN SPIKE=<*signals*>

triggers a warning when a spike hazard occurs at any of the specified signals, while:

WARN SPIKE:

triggers a warning when a spike hazard occurs at any signal.

The **STROBE** keyword specifies the condition that a **strobe error** occurs at any primary output or bus. This is only defined for tester emulation mode. The command:

WARN STROBE:

triggers a warning for this situation.

The **UNSTABLE** keyword designates the condition that a change of primary input state has occurred while the circuit was still in an unstable state (loss of fundamental mode of operation for waveform and tester emulation modes). The command:

WARN UNSTABLE:

triggers a warning for this situation.

Warn On The Occurrence Of Timing Check Violations

Warning messages can be requested for timing check violations at SIMIC latches and flip-flops. The **PARTS** keyword is used to specify which memory elements to monitor for the selected timing check(s). The command:

WARN PARTS=<*parts_ff*> <*timing-check*>: ...

selects specific memory elements to monitor. For example,

WARN PART=ff1,ff2 SETUP.D: PW:

triggers warning messages for data setup-time and all pulse-width check violations at the memory elements named **ff1** and **ff2**.

The command:

WARN PARTS: <*timing-check*>: ...

triggers warning messages when violations of the specified timing checks occur at any memory element. For example,

WARN PART: SETUP: HOLD: PW:

will cause any timing check violation at any memory element to trigger a warning message.

Warn On The Occurrence Of Wire-tie Conflicts

The **CONFLICT** keyword specifies a wire-tie conflict. The command:
    WARN CONFLICT=<*signals*>
triggers a warning when a wire-tie conflict occurs at any of the specified signals, while the command:
    WARN CONFLICT:
triggers a warning when any wire-tie conflict occurs.

The **CONFLICT** keyword accepts the same type of signal specification as the **LIST** keyword.

Warn On Oscillation

The **OSCILLATION** keyword specifies excessive activity in response to a single change of input state (see also **DEFINE OSCILLATION**). The command:
    WARN OSCILLATION:
triggers a warning when any signal exhibits excessive activity.

Suppressing Excessive Messages On A Per-Signal Basis

The **STOP** keyword option controls the maximum number of per-condition warning messages SIMIC issues for each signal. The command form is:
    WARN STOP=<*integer*>
By default, SIMIC will only display 10 SPIKE messages, 10 PULSE messages, etc., for each signal. Thus, SIMIC's default is equivalent to:
    WARN STOP=10
The **STOP** limit must be an integer ranging from 1 to 511.

To prevent any messages from being suppressed, use the command:
    NO WARN STOP:

Directing Warn Messages

By default, warning messages are directed to the terminal. The command:
    NO WARN TERM:
disables warn messages at the terminal. These messages can be re-enabled at the terminal with the command:
    WARN TERM:

Warn messages can also be directed a file with the **FILE** keyword. The form:
    WARN FILE=<*file name*>
explicitly specifies a file name, and possibly its extension. The command:
    WARN FILE:

specifies a file with the default file name (see **DEFINE FILE**). If the second form is used, or if no file extension is specified in the first form, the file's default extension is **wrn**.

Regardless of how they were enabled, warn messages to a file are disabled with the command:

```
NO WARN FILE:
```

Run Command: WRITE (WR)

Function

The **WRITE** command is used to report selected signal values in a tabular format during simulation. Output is directed to a file.

Usage

The **WRITE** command is described in Chapter 2.4.

The **WRITE** command can be restricted to a specified interval of tests, for patterns, or time, for waveforms, with the **PRANGE** keyword option:

`WRITE PRANGE=`*<prange spec>*

Specifying The File Name For The WRITE File

By default, the extension of the file created by the **WRITE** command is **wrt**, and its name is the default name specified by the **DEFINE FILE** command. The **FILE** keyword can be used to explicitly specify this file's name:

`WRITE FILE=`*<file name>*

where ***<file name>*** is the name of the file to be written to.

Specifying Signals and Format

The **LIST** keyword is used to select signals and format the output: The command form for this selection is:

`WRITE LIST=`*<signals and format>*

where ***<signals and format>*** has the form of a ***<signals>*** specification, possibly augmented with formatting options. These options are:

1. Inserting one or more blank vertical columns between signals. This is accomplished by entering an asterisk (*****) for each blank column.
2. Forcing a new row in the table. This is accomplished by entering a pound sign (**#**) at the desired point.

Commas or whitespace are optional before or after ***** or **#**.

For example:

`WRITE LIST=u,v,*w**x#y`

will cause the simulation output to consist of the value of signal **u**, followed by the value of **v**, followed by a blank column, followed by the value of **w**, followed by two blank columns, followed by the value of **x**. The value of signal **y** will be output at the first position of the next line.

The second form of the **LIST** keyword:

```
WRITE LIST:
```

specifies that the values of *all* signals be reported in “dump” format; groups of five values ordered alphabetically, according to the **symbol** section of the listing file.

The **WRITE** command’s **LIST** keyword is “sticky”; any signal specified will continue to be reported until explicitly removed. Removing signals from the list is accomplished with the **NO** prefix. For example:

```
NO WRITE LIST=a,b
```

removes signals **a** and **b** from the list of reported signals. The format options “*” and “#” are not specified with the **NO** prefix. The command:

```
NO WRITE LIST:
```

removes all signals, thereby terminating **WRITE** output.

Suppressing Signal Strength

By default, a 15-character set is used to represent combined level and strength information. Optionally, output can be restricted to a 4-character set representing levels; **0**, **1**, **X**, and **Z** (although **Z** represents the combination of unknown level at floating strength). The **VALUES** keyword option is used to select value representation. The command:

```
WRITE VALUES=LEVELS
```

selects the 4-character representation, while the command:

```
WRITE VALUES=STRENGTHS
```

restores the 15-character default representation.

Requesting Output At Stable Points

The default **WRITE** operation is to output the requested simulation values each time the circuit becomes stable. The **PSTEP** (**PS**) keyword option can be used to control frequency of this output, or inhibit it. The command:

WRITE PSTEP=<*integer*>

changes this interval. For example:

```
WRITE PSTEP=5
```

specifies that output occur every fifth stable point.

Output based on attaining a stable state can be disabled with the **NO** command prefix:

```
NO WRITE PSTEP:
```

Requesting Time-Periodic Output

Output can be requested at specified time intervals using the **TSTEP** keyword. The command:

WRITE TSTEP=<*integer*>

will cause an output to occur every ***<integer>*** simulation time-units. If the time-step specification is preceded by a plus (“+”) sign, output will continue regardless of circuit stability. For example:

```
WRITE TSTEP=+100
```

produces output every 100 time-units, even when the circuit state is stable.

The **NO** prefix can be used to disable time-periodic output:

```
NO WRITE TSTEP:
```

The **BEGIN** keyword option may be used to skew (offset) the first **TSTEP** output. This command has the form:

WRITE BEGIN=*<integer>*

where ***<integer>*** specifies the time offset to the first output.

To remove the offset, specify a value of 0:

```
WRITE BEGIN=0
```

Requesting Output Based On Activity

The **CHANGE** keyword option specifies that output be triggered by activity at monitored signals. The command form is:

WRITE CHANGE=*<signals>*

where ***<signals>*** is a subset of the signals currently being monitored with the **WRITE** command (other signals are ignored). This command specifies that output should occur whenever any of the specified signals changes state.

The command:

```
WRITE CHANGE:
```

will trigger write output when any of the monitored signals changes state.

Subsequently, the command:

NO WRITE CHANGE=*<signals>*

would inhibit the specified signals in ***<signals>*** from functioning as triggers, while:

```
NO WRITE CHANGE:
```

would disable all activity-based output.

Header And Tabular Format Control

SIMIC, by default, limits each output line to 80 columns. The **EXPAND** keyword option can be used to modify output line width. The command form:

```
WRITE EXPAND:
```

expands output line width to 132 columns, while the command form:

```
WRITE EXPAND=INFINITE
```

allows arbitrary output line width.

The **HEADER** keyword option controls whether the simulation header, consisting of the simulation options and signal names, be output prior to the tabular simulation values. By default, the header is enabled. The command:

```
NO WRITE HEADER:
```

disables the header output, and the command:

```
WRITE HEADER:
```

enables it.

When waveform stimuli are used, the test number field of the simulation output can be optionally suppressed with the **TNUM** keyword option. By default, this field is present in every record. The command:

```
NO WRITE TNUM:
```

suppresses the test field. Each output line will contain the time, followed by a colon, followed by the requested signal values.

The test field can be restored with the command:

```
WRITE TNUM:
```

Run Command: XPROPAGATE (XP)

Function

The **XPROPAGATE** command is used to globally enable or disable X-pulse creation/propagation for **spike hazards** and **near hazards**.

The **XPROPAGATE** command is used to selectively enable or disable X-pulse creation/propagation for timing check violations at SIMIC D latches and edge-triggered flip-flop primitives.

The **XPROPAGATE** command is used to selectively modify the **filter** and **liberal** spike control parameters.

Usage

SIMIC, by default, propagates an X whenever a spike, or part timing violation (setup, hold, or pulse-width) occurs. X-propagation is controlled by the **XPROPAGATE** command.

Globally Enabling And Disabling X-Propagation For Hazards

The **SPIKE** keyword option globally controls whether X-propagation is performed for spike hazards. The command:

```
NO XPROPAGATE SPIKE:
```

globally disables X-propagation for spike hazards, while the command:

```
XPROPAGATE SPIKE:
```

globally re-enables it.

The NEAR keyword option globally controls whether X-propagation performed for near hazards. The command:

```
XPROPAGATE NEAR:
```

enables X-propagation for near hazards, while the command:

```
NO XPROPAGATE NEAR:
```

disables it.

By default, X-propagation is disabled for near hazards.

Controlling X-Propagation For Functional Timing Violations

The **PART** keyword option, combined with individual timing check specifications, allow X-propagation for timing check violations to be controlled on a per-instance, per-check basis. The command:

NO XPROPAGATE PARTS=<*parts_ff*> <*timing-check*>:

disables X-propagation for the specified timing check violations and parts, while the command:

`NO XPROPAGATE PARTS: `*<timing-check>*`:`

disables X-propagation for the specified timing checks for all parts. Different timing checks may be specified in the same command.

For example, assuming part **`f1`** is a DCF primitive, the command:

`NO XPROPAGATE PART=f1 HOLD: SETUP.NR:`

would disable X-propagation for all hold-time violations and for setup-time violations between the **`NR`** and **`CLK`** pins at this flip-flop.

Subsequently, X-propagation can be re-enabled with the same commands, except the **`NO`** prefix would be omitted. For example:

`XPROPAGATE PART=f1 HOLD: SETUP.NR:`

re-enables X-propagation for all hold-time violations and for setup-time violations between the **`NR`** and **`CLK`** pins at flip-flop **`f1`** of the previous example.

Modifying Spike Control Parameters

The **liberal** and **filter** spike control parameters can be modified independently with the **`XPROPAGATE`** run command. The commands:

`XPROPAGATE FILTER=`*<filt_lib>*` LIST=`*<signals>*

`XPROPAGATE LIBERAL=`*<filt_lib>*` LIST=`*<signals>*

change the **filter** and **liberal** parameters, respectively, to the value ***<filt_lib>*** for the selected signals in ***<signals>***. The commands:

`XPROPAGATE FILTER=`*<filt_lib>*` LIST:`

`XPROPAGATE LIBERAL=`*<filt_lib>*` LIST:`

change the **filter** and **liberal** parameters, respectively, for all signals.

The ***<filt_lib>*** value is a percentage from 0 to 100, and is specified as an integer, optionally followed by a percentage sign (e.g. 50 or 50%). For example,

`XPROPAGATE FILTER=20% LIST=abc,def`

sets the **filter** spike control parameter of signals **`abc`** and **`def`** to 20%.

An asterisk (**`*`**) may be also be specified for ***<filt_lib>*** to restore the spike control parameters to their original values (in the SNL description). For example:

`XPROPAGATE LIBERAL=* FILTER=* LIST:`

would reset all **`FILTER`** and **`LIBERAL`** parameters to their original values.

Index