ValidSIM™ REFERENCE MANUAL

Manual Number: MN224 Rev.A

15 July 1986

Valid Logic Systems, Incorporated
2820 Orchard Parkway
San Jose, CA 95134
(408)945-9400 Telex 371 9004
FAX 408 262 2599

MANUAL REVISION HISTORY

Rev	Date	Software Release	Reason for Change
A	7-15-86	ValidSIM Release 2.1	Initial release.

TABLE OF CONTENTS

Section 1
Introduction
Theory of Operations 1-2
Signal States 1-2
Bidirectional Nets 1-4
Combination of States 1-4
Getting Started 1-6
Selecting the Appropriate Simulation 1-6
Window Sizes 1-7
Directives Files 1-7
Starting Simulation 1-8
The Simulator Display 1-9
Status Lines 1-10
Echo Area 1-11
Display Area 1-12
Display Modes 1-12
Displaying Signal Values 1-13
Signal Values in Bus Mode 1-14
Signal Values in Waveforms Mode 1-14
Graphics for Signal Values 1-14
Using Character Graphics 1-16
Working with Signal Values 1-17
Opening Signals 1-17
Opening Memories 1-18
Initialization of Signals and Memories 1-18
Changing Signal Values 1-19
Changing Memory Values 1-19
Advancing Simulation Time 1-19
Simulator Output 1-20
Listing File 1-20
Log File 1-21
Command File 1-21
Waveform Input File 1-21
File Names 1-21
Split-Screen Simulation 1-22
Differences from Full-Screen Simulation 1-22

Section 2
Logic Simulator Directives

Section 3
Waveforms Mode
Waveform Commands 3-2

Section 4
Breakpoints and Patching
Breakpoints 4-1
Breakpoint Commands 4-2
Combining Breakpoint Commands 4-5
Expression Syntax 4-6
Logic Patching 4-7

Section 5
Tracing and Tabular I/O
Tracing 5-1
Requirements for Standard Tracing 5-1
Signal Mapping 5-2
Value Information for Tracing 5-2
File Formats for Tracing 5-2
Signal Mapping File Format 5-3
Value File Format 5-5
Tabular I/O 5-6
File Format for Tabular I/O 5-6
Stimulating Circuits with Tabular I/O Files 5-7
Sample Tabular I/O Use 5-8

Section 6
Loading Memories

Section 7
Simulator Commands

Section 8
Delays

Delay Properties 8-1
Delay Property 8-1
Rise and Fall Properties 8-2
Pin-to-Pin Delay 8-2
Rise_Fall Directive 8-3
Examples Using Delay Properties 8-4
Pin Delays with Rise/Fall Delays 8-4
Pin Delays without Rise/Fall Delays 8-5
Delay Estimator 8-6
Interaction of Wire Delays with Delay Estimator 8-7
Computing Net Dependent Delays 8-7
Using the Delay Estimator 8-10
Expression Evaluator 8-12
Wire Delay Feedback 8-15
Wire Delay File 8-15
Using a Wire Delay File 8-17

Section 9
Simulation Models

Using Simulation Primitives 9-1
Simulator Primitives 9-3
Logic Gate Primitives 9-3
Buffer Primitives 9-5
JK Primitive 9-6
Latch Primitives 9-7
Register Primitives 9-9
Multiplexer Primitives 9-13
Memory Primitive 9-14
Counter/Shift Register Primitive 9-16
Arithmetic Primitives 9-18
Timing Checker Primitives 9-20
Encoder and Decoder Primitives 9-23
Other Primitives 9-24
User-Coded Primitives 9-25
Properties Affecting Simulation 9-25

Section 10
Error Messages

Appendix A
S-32/S-320 Additional Features

Save and Restore Function A-1
Save Command A-1
Restore Command A-2
Restore Directive A-2
Simulator Interruption A-2
User-Coded Primitives A-3
The Pascal Code for Systems Running UNIX A-4
The Pascal Code for Systems Running VMS A-5
The Pascal Code for Systems Running CMS A-6
Running a Simulator Containing UCPs A-8
Running Your Simulator Under UNIX A-8
Running Your Simulator Under VMS A-8
Running Your Simulator Under CMS A-9
Body Definition for UCPs A-9
UCP Pinout Descriptions A-10
User_Prim_Config Directive A-13
Own Storage in UCPs A-13
Functions Provided for Use in UCPs A-13
Example of a User-Coded Primitive A-18
User Configuration File A-19
VMS Pascal Module Example A-20

Appendix B
PC AT Additional Features

Save and Restore Function B-1
Save Command B-1
Restore Command B-2
Restore Directive B-2

Index

SECTION 1
INTRODUCTION

The Logic Simulator represents a new approach to simulation of large digital systems. By separating timing verification from simulation, timing verification has been made more comprehensive and simulation has been made conceptually simpler and thus much faster. Optional timing analysis features in the Simulator also allow the user to perform some timing verification if desired.

First the design is entered using GED the Graphics Editor and the signals are named using the SCALD language. Buses can be drawn as a single wire by including a bit subscript in the signal name. The SIZE and TIMES properties can be used to let a one-bit section represent a multibit-wide part. After the design is entered, you call the Simulator. The Simulator calls the Compiler to expand the design using the simulation models for each library part used in the design. The Compiler works on a design a page at-a-time so that when a small change is made to a design, only a single page needs to be recompiled. For special purposes, the Compiler may also be called directly and a single large expansion file may be produced for a design. See the Compiler Reference manual for details.

Although designed primarily as an interactive tool, the Logic Simulator may also be run as a batch process. All commands used during a simulation session may be entered into a single command file and the COMMAND_FILE directive may be entered in the Simulator Directives file to direct the Simulator to use the commands in the specified file.

The designer may use command files to exercise a design in a way that is analogous to a diagnostic program. Since command files may be stored for repeated use, verification of a previously checked circuit can easily be repeated to

ensure that it is still working correctly after design modifications.

1.1 THEORY OF OPERATIONS

The Logic Simulator initializes the system to a fixed state and waits for a command from the user. The user enters commands interactively (or through a command file) to OPEN the signals that need to be tracked, to DEPOSIT initial values to some signals, and to advance simulated time. It is frequently necessary to provide an external stimulus to a design, for example, a simulated disk data stream. Application of stimulus may be done through a command file, data file, or in many cases, by simulating an additional circuit specifically drawn to provide the stimulus.

After each SIMULATE command, the Simulator reports signal values for all OPENed signals. In Bus Mode simulation (useful for very large designs) instantaneous signal values are given for each signal. A history of previous signal values is not kept.

In Waveform Mode simulation, the signal value of each signal is recorded as a waveform. The previous signal values for each OPENed signal are kept for the amount of simulated time specified with the HISTORY command.

SIGNAL STATES

Each bit of each signal in the Simulator assumes one of the 20 internal signal states used by the Simulator. These 20 internal states are mapped into 12 states that are used to report signal values to the user. The eight states that are not used to report signal values are special states that are used internally.

Each state is made up of two parts, a VALUE and a STRENGTH. The VALUE of a signal is its logical level. There are three possible signal values. These are:

Signal Value	Meaning
0	Logical 0
1	Logical 1
U	Unknown (could be 0 or 1)

The STRENGTH of a signal describes the type of output or outputs that drive the signal to its VALUE. The possible STRENGTHs are:

Signal Strength	Meaning
HARD	Driven to level without resistance
SOFT	Driven to level through resistance
MEMORY	Driven to level and holding (due to charge storage)
INDETERMINATE	Could be HARD, SOFT or MEMORY

MEMORY STRENGTH signals maintain their VALUE for the period of time specified with the DECAY_TIME directive. After this time they assume UNDEFINED VALUE. DECAY_TIME is measured from the time the signal was last driven to the specified value. All signals in a design have the same decay time. The default value for DECAY_TIME is infinite.

The combination of each of the three signal values with each of the four signal strengths gives the 12 signal states that are reported. The combination of INDETERMINATE strength and UNKNOWN value is interpreted as Z (high-impedance). The state names and abbreviations are:

STATE NAME	ABBREVIATION
HARD_STATE_0	h0
SOFT_STATE_0	s0
MEMORY_STATE_0	m0
INDETERMINATE_STATE_0	i0
HARD_STATE_1	h1
SOFT_STATE_1	s1
MEMORY_STATE_1	m1
INDETERMINATE_STATE_1	i1
HARD_STATE_U	hU
SOFT_STATE_U	sU
MEMORY_STATE_U	mU
STATE_Z	Z

BIDIRECTIONAL NETS

Bidirectional nets are nets that connect to the pins of a PASS TRANSISTOR or RES primitive. DEPOSITing into bidirectional signals is not recommended as the deposited value does not persist very long due to the bidirectional net evaluation scheme used by the Simulator. Unidirectional drivers should be connected to those bidirectional nets that the user wishes to force to certain levels.

COMBINATION OF STATES

When more than one output drives a net, the state of the net is determined by combining the states of the driving outputs. When more than two outputs drive a net, the output states are combined iteratively. The following tables list all combinations of two states.

	h0	s0	m0	i0	h1	s1
h0	h0	h0	h0	h0	hU	h0
s0	h0	s0	s0	i0	h1	sU
m0	h0	s0	m0	i0	h1	s1
i0	h0	i0	i0	i0	hU	hU
h1	hU	h1	h1	hU	h1	h1
s1	h0	sU	s1	hU	h1	s1
m1	h0	s0	mU	hU	h1	s1
i1	hU	hU	hU	hU	h1	i1
hU	hU	hU	hU	hU	hU	hU
sU	h0	sU	sU	hU	h1	sU
mU	h0	s0	mU	hU	h1	s1
Z	h0	s0	m0	i0	h1	s1

	m1	i1	hU	sU	mU	Z
h0	h0	hU	hU	h0	h0	h0
s0	s0	hU	hU	sU	s0	s0
m0	mU	hU	hU	sU	mU	m0
i0	hU	hU	hU	hU	hU	i0
h1	h1	h1	hU	h1	h1	h1
s1	s1	i1	hU	sU	s1	s1
m1	m1	i1	hU	sU	mU	m1
i1	i1	i1	hU	hU	hU	i1
hU	hU	hU	hU	hU	hU	hU
sU	sU	hU	hU	sU	sU	sU
mU	mU	hU	hU	sU	mU	mU
Z	hm1	i1	hU	sU	mU	Z

1.2 GETTING STARTED

This section describes the different ways you can invoke the Simulator: the split-screen Simulator and the full-screen Simulator. It also describes window sizes and explains the two methods of invoking the Compiler.

SELECTING THE APPROPRIATE SIMULATION

The Simulator can be run on any of Valid's supported hardware configurations.

The two most common ways to run the Simulator are in full screen simulation with graphics or under GED in split-screen simulation. Full screen simulation is used for medium to large designs because signal values for 48 signals can be viewed simultaneously on the display. Split-screen simulation is useful for smaller designs and for opening signals. You enter split-screen simulation directly from GED and the schematic appears in the top portion of the screen while the lower portion is used for the simulation display. Signals can be specified by a puck press in the GED window, rather than typing a signal name. All of the GED commands are available to the user as well as all of the Simulator commands. Because part of the screen shows the schematic in split-screen simulation, a fairly large screen is required and only 12 signals can be viewed at one time in the simulation display (use the ROW command to see the others). See Split Screen Simulation later in this section for additional information.

In addition to these two methods of running simulations, the Logic Simulator can also be run on an IBM or VAX mainframe. Full screen simulation is available in both Waveforms and Bus mode. All commands that do not require a puck are available, and in Waveforms mode, waveforms appear in character-graphics (because of the limitations of the hosts). User-coded primitives (UCPs) may also be used during simulation on an IBM or VAX mainframe.

WINDOW SIZES

If the Simulator is invoked in a partial screen window, the window must be at least a minimum size. Under GED, this size is 48 x 86 characters. The Simulator with character-graphics can be run in any window at least 12 x 80; with regular graphics, the minimum window size is 22 x 80. The user is prevented from running the Simulator in any window that is smaller than required.

DIRECTIVES FILES

First, use the Graphics Editor to create the design to be simulated. Then edit the Compiler Directives file. Here is an example of a Compiler Directives file:

```
root_drawing 'subtractor';
compile sim;
library standard, sim, lsttl, tutorial;
directory 'susan.wrk';
directory '/u0/lib/mylib/mylib.lib';
warnings on;
oversights on;
output list, expand, synonym;
print_width 80;
suppress 196;
end.
```

In this sample directives file, the drawing to be compiled is named "subtractor." This drawing resides in the SCALD directory "susan.wrk". The format of the Compiler directives file is the same whether the Compiler is called directly (with the **compile** command), or from within the Simulator (**simulate** command). Any errors reported during compilation must be corrected before the design is simulated.

Next, edit the Simulator Directives File. Include the ROOT_DRAWING directive and any other required directives. Specify the same root_drawing name as in the Compiler directives file. When the ROOT_DRAWING directive is included, the Simulator can call the Compiler directly.

Directives may be entered in either upper or lower case. Comments may be included if enclosed in curly brackets. Each directive must be terminated with a semicolon (;) and the file must end with an **END.** statement. Here is a sample Simulator Directives File.

```
ROOT_DRAWING 'subtractor';
CLOCK_PERIOD 100;
CLOCK_INTERVALS 5;
OUTPUT LIST, command_log;
SESSION_LOG ON;
TERMINAL GRAPHICS;
USE_IF BATCH; { rest of directives for batch only }
COMMAND_FILE 'BATCH.CMD';
TERMINAL TTY; { terminal type for batch mode }
END.
```

When the Simulator Directives file is correct, the Simulator can be invoked immediately. Type the command "simulate" at the system prompt.

STARTING SIMULATION

The Simulator starts by reading the Simulator directives file (simulate.cmd). If the ROOT_DRAWING directive is specified in this file, or the drawing name given on the simulate command line, the Simulator calls the Compiler directly if it needs to, and then starts the Simulator if no Compiler errors are found. A command line argument overrides any root drawing name in the directives file. The root drawing name should match that in the Compiler directives file.

When the ROOT_DRAWING directive is used, the COMPILER_OUTPUT and SYNONYM_FILE directives are ignored and any existing Compiler expansion and synonyms files are also ignored. The Compiler does NOT generate the expansion and synonyms files.

The Compiler generates error messages if there are any errors during the compilation. If the specified root drawing name is not found or if compile errors are detected, the Simulator will exit and Comperr is run automatically to collect the error reports. Look in the Compiler listing file (cmplst.dat) for Compiler error messages.

The program COMPERR can also be invoked explicitly to collect all the Compiler error messages. Any errors reported during compilation must be corrected before the design is simulated.

When there are no Compiler errors, the Simulator display appears.

If the ROOT_DRAWING directive is not used, the Simulator requires as input two Compiler output files: the expansion file (cmpexp.dat) and the synonym file (cmpsyn.dat). If the names of these files are not the default names, use the COMPILER_OUTPUT and SYNONYM_FILE directives to specify the file names. If, for backwards compatibility, you need to generate Compiler output files, see the Compiler Reference Manual.

1.3 THE SIMULATOR DISPLAY

The Simulator display screen is divided into three parts: the status lines at the top, the display area in the middle, and the echo area, at the bottom.

- The status lines report current simulation parameters such as the current simulation time, the current radix (or base) and the current scale or resolution. The status lines are updated periodically by the Simulator.

- The display area shows the values of signals selected by the user and, in Waveforms mode, their history.

- The echo area echos the commands as they are selected from the menu, as they are typed from the keyboard, and as they are issued from a script file.

On platforms having multiple windows, the size of the Simulator display area in full-screen simulation varies with the size of the window in which the Simulator is invoked. Not only does the number of lines increase (for up to 48 waveforms in a full-screen window), but the width of the display area also grows as the size of the window is increased above the minimum. The additional width is used to increase the space available for waveforms in Waveforms mode and to increase the total amount of space available for signal names and values in Bus mode.

STATUS LINES

The status lines display the following fields:

Time	Radix
Clock	Top Row
Scale	Mem path
Scope	

Mem path and **Scope** are displayed only in Bus mode, and **Top Row** is displayed only in Waveforms mode. **Scope** appears in the status line in Bus mode and below the display field in Waveforms mode.

- **Time** is the current simulation time in nanoseconds.

- **Radix** shows the current radix value, which can be binary, octal, decimal, hexadecimal, or strength.

- **Clock** indicates the clock period in nanoseconds and the number of intervals into which the clock has been subdivided.

- **Top Row** indicates the row number of the top row on the display in Waveforms mode.

- **Scale** is the scale factor that is set with the RESOLUTION directive.

- **Mem path** is a memory pathname that indicates the result of the last successful MEMPATH command.

- **Scope** shows the default path name that may be set with the SCOPE command.

ECHO AREA

The echo area is used for echoing command inputs and displaying Simulator output information. Most of this information consists of either error messages or query responses.

In Waveforms mode, the first line of the echo area has three fields: Trigger, Cursor, and Scale.

- **Trigger** is the time the most recent Breakpoint was encountered.

- **Cursor** shows the location of the cursor. The cursor advances with simulation and can also be moved with the CURSOR command. The CURSOR command is useful for determining the time of a transition. The cursor may be placed at any time between 0 and the current time. The values of the signals on the display at the time specified by the cursor are displayed on the right side of the screen.

- **Scale** is the Scale factor that is set with the RESOLUTION directive.

When running under GED, if a command results in several lines of output, the Simulator displays a few lines and prints

** Press <RETURN> to continue **

The Simulator then waits for a < RETURN > and ignores all other input until a < RETURN > is entered.

DISPLAY AREA

In Waveforms mode a time line appears across the bottom of the display area and matching tick marks across the top. The characters **T**, **C**, and **B** also appear along the top of the display area. The **T** character marks the time of the most recent Trigger. The **C** character marks the cursor. When the **T** and **C** are in the same location, a **B** (for both) is displayed.

DISPLAY MODES

The Simulator has two display modes. These are BUS mode and WAVEFORMS mode. Bus mode displays instantaneous values of a large group of signals selected by the user. Bus mode is used for very large circuits. Waveforms mode displays signal values for each signal as a waveform over time. Signals (scalars or buses) are displayed one to a line. In Waveforms mode signal values for up to 200 signals can be maintained. The number of signals that can be simultaneously displayed on the screen is determined by the type of terminal, as shown in the following table.

Terminal Type	Maximum Number of Waveforms Displayed
S-320/S-32	48
IBM PC AT/GX	48
IBM PC AT/EG	17
IBM PC AT/VG	42
MicroVAX	48
Ann Arbor	34
VT100	12
IBM 3270	14

In the Waveforms mode, both the current value and a history of transitions are maintained for each signal. The signal history is displayed as a waveform (similar to that produced by a standard logic analyzer) and can be written to a file (using the PLOT command) for subsequent input to the Plottime program. The Plottime program produces timing diagrams that are GED drawings. See the Plottime Reference Manual.

1.4 DISPLAYING SIGNAL VALUES

Signal values are displayed in one of five radices: binary, octal, decimal, hexadecimal, or strength. Binary numbers are indicated by a trailing "b", octal by "o", decimal by "d", hex by "h", and strength by "s". On input, numbers are assumed to be in the current radix. Each bit of a value may be either 1, 0, U (unknown), or Z (high impedance).

In binary, these bits are displayed as "1", "0", "U" or "Z". Unknown or high-impedance bits in octal or hex values cause the digit to which they map to be reported as "Z" if all bits in the digit are high impedance, otherwise "U". Unknown or high-impedance bits in decimal values cause the entire value to be displayed as "U".

In strength radix, values are displayed and input using the state abbreviations shown above (page 1-4). For example, "U0010ZZZZb" represents a binary value with the most significant bit unknown and the four least significant bits in high impedance. In hex, this value would be displayed as "U2Zh". In strength radix, this value might be displayed as ".hU.hO.s0.s1.mO..Z..Z..Z..Zs" if some of the bits were HARD, some SOFT, and some MEMORY strength. In Waveforms mode, "---------" is output in the column of signal values if there is insufficient space to display the entire value.

SIGNAL VALUES IN BUS MODE

In Bus mode many signals are displayed on each line. A signal might not fit if its name and value require too much room horizontally. If this happens, open the signal instead as several subranges. The value for each range of bits will be shorter than the total signal value.

SIGNAL VALUES IN WAVEFORMS MODE

Waveforms mode uses graphics to display signal values whenever the terminal has graphics capabilities. When running simulation on a VAX or IBM mainframe, where graphics are not available, character graphics are used. On the SCALDSystem IV waveforms look slightly different than on other systems because they are formed using a graphical character set.

GRAPHICS FOR SIGNAL VALUES

When standard graphics are being used, single bit signals (scalars) are represented as shown in Figure 1-1:

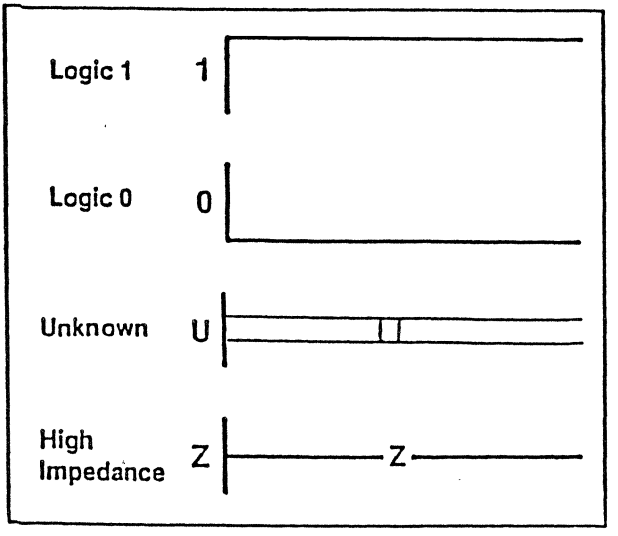

Figure 1-1. Graphics for Scalar Signals

Multiple bit signals (buses) are represented as shown in Figure 1-2:

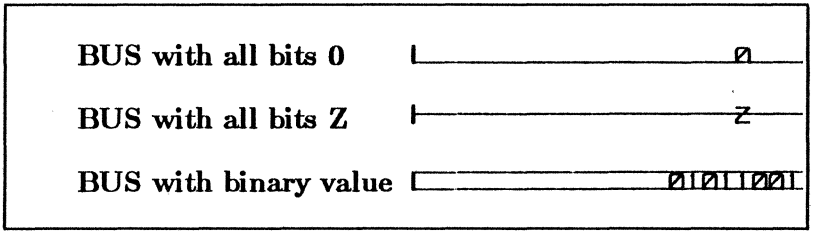

Figure 1-2. Graphics for Bus Signals

When using standard graphics, all signal transitions are indicated by a vertical line at the transition time; multiple transitions at a single time are indicated by a bold vertical line.

A typical scalar signal is shown in Figure 1-3:

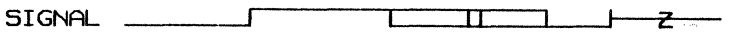

Figure 1-3. Scalar Signal

This signal has the history: 0,1,U,0,Z.

A typical bus signal (vector) is shown in Figure 1-4:

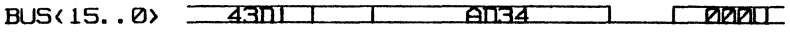

Figure 1-4. Bus Signal

This bus has the history: 43D1,?,AD34,0000,000U (where "?" indicates that the value is too large to fit in the display space). To view a signal value that is too large to fit in the display, use the WAVEFORM command to zoom in on the simulation time when that signal value occurred.

USING CHARACTER GRAPHICS

When character graphics are being used single bit signals (scalars) are represented as follows:

Logic 1	^^^^^^^
Logic 0	_____
Unknown	==U==
High Impedance	---Z---

Multiple bit signals (buses) are represented as follows:

All bits 1	^^*val*^^
All bits 0	___0___
All bits Z	---Z---
With value *xxx*	==*xxx*==
With value too large to display	=======

When character-graphics are being used, the following characters are used to indicate different transitions:

/	low-to-high transition
\	high-to-low transition
X	multiple transitions mapping to same character
>	transition to Z
|	all other transitions (including transition to U)

A multiple transition symbol is displayed only if some bit of the signal has changed two or more times during the time mapping to the display position. If a bus undergoes several transitions, but each bit changes only once, a multiple transition has not occurred.

1.5 WORKING WITH SIGNAL VALUES

OPENING SIGNALS

When the user "opens" a signal name, that signal and its value appear in the main display. Signal names are in standard SCALD syntax, except that bit lists and step values are not permitted. Names are right-justified in Waveforms mode. The currently open signal is indicated by a "->" preceding the signal value. The value of the signal appears left-justified in the current radix. This value may be changed by "depositing" some other value. The last signal to be opened may be changed at any time in this fashion.

Any subrange of a signal may be displayed. If no bit range is given for a signal vector, then the entire vector is OPENed. A signal also may be displayed any number of times in different radices. A change made to the value of one version of the signal affects all the others.

A signal may be known by more than one name. A bit may be common to a group of signals. For example, the signal ADR BUS<31..0> may also be known as SYSTEM BUS<71..40> and also as CHIP SELECT<5..0>: MEMORY ADR<27..0>. The user may refer to the signal using any of its names.

OPENING MEMORIES

The contents of memories are displayed in a somewhat different fashion from signals. First, the pathname to the memory primitive is specified with the MEMPATH command (see the Commands section) and then the OPEN-MEMORY command is used to open the desired addresses. This sequence causes an entry to be made in the main display area. This entry displays on the left the memory pathname followed by the address (enclosed in parentheses). On the right the value stored in the memory at that address is displayed (right justified in the current radix). Bit ranges are not permitted in memory displays.

INITIALIZATION OF SIGNALS AND MEMORIES

At the start of a simulation session, each signal is set to the undefined state U. The LOGIC_INIT and MEM_INIT commands are used to initialize all signals or all memories to a specific value. Signals and memories may be initialized to 0, 1, undefined (U), asserted (*) or unasserted (-*). If a signal has neither low assertion nor negation characters, or if it has both, then its asserted state is one; otherwise it is zero. For example, if - is the negation character and * is the trailing low assertion character, then SIG A and -SIG B* have an asserted value of one, while -SIG C and SIG D* have an asserted state of zero. Memories with a bubbled output have an asserted state of zero; other memories have an asserted state of one.

CHANGING SIGNAL VALUES

The user may change the value of any signal, whether driven or undriven. When the user changes the value of a signal that is driven, the signal value specified remains on the net until overridden by a driver. When a signal has a clock assertion (!P or !C) and is not driven by an output in the circuit, the signal generated by the Logic Simulator has the timing behavior specified by the clock assertion. Signal assertions with the prefixes !S and !D are ignored by the Simulator.

CHANGING MEMORY VALUES

The user may change the value of any memory location in the design by opening a location with the OPENMEMORY command and then DEPOSITing the desired value to that location. The location retains this value until a memory write is done to the matching address or until another DEPOSIT is done into that location.

1.6 ADVANCING SIMULATION TIME

The user can cause the simulated time to advance by typing "SIMULATE C" (simulate for a clock period), "SIMULATE S" (simulate for the STEP time), or "SIMULATE *value* (simulate for *value* nanoseconds). Primitives are evaluated and values are changed as time advances until the designated simulation time elapses. When this time is reached, the status lines and the values of all the signals in the main display are updated appropriately. The command "SIMULATE 0" can be used to immediately cause the evaluation of any zero-delay parts.

The Simulator is able to detect zero-delay loops during simulation (except when using Realfast). This facility does not attempt to detect all the possible oscillations, but simply the loops that would otherwise cause the Simulator to enter an infinite loop. When a zero-delay loop is detected, a warning message appears and the SIMULATE command is terminated. You will typically want to exit from the Simulator and fix the error in the design.

1.7 SIMULATOR OUTPUT

The Simulator produces four output files: a listing file (simlst.dat), a log file (simlog.dat), a command file (simcmd.dat), and a Waveform Input file (plotsig.dat).

File names for these files are slightly different when the Valid verification tools are run under different operating systems (UNIX, VMS, CMS). Under UNIX, the file names are:

Simulator Listing File	=	**simlst.dat**
Simulator Log File	=	**simlog.dat**
Simulator Command File	=	**simcmd.dat**
Waveform Input File	=	**plotsig.dat**

For file names under VMS and CMS, see File Names later in this section.

The log file is produced every time you run the Simulator. The OUTPUT directive regulates whether the listing file and the command file are produced. The Waveform Input file is produced when you use the PLOT command.

Each time the Logic Simulator is run from the same directory, the output files are overwritten. This saves considerable amounts of disk space. Here is a brief description of each of the Simulator output files.

LISTING FILE

The Listing file **simlst.dat** contains a summary of the directives, process information on the Simulator run, and error information. The directive SESSION_LOG ON causes a record of all terminal I/O for the Simulator session to be sent to the listing file. The SNAPSHOT command sends an image of the status lines and signal display window to the listing file. By using the SESSION_LOG directive and the SNAPSHOT command a permanent record of a simulation session can be created. For more details, see the section on Logic Simulator Directives.

LOG FILE

The Log file **simlog.dat** is used primarily by internal personnel to track down Simulator errors. In addition to process and error information, this file contains statistics on the memory requirements of the Simulator run.

COMMAND FILE

The Command file **simcmd.dat** is a list of all of the commands issued in a Simulator session. It is used to exactly duplicate a simulation session, or to run a batch simulation. Edit the simcmd.dat file as required and rename it. Then use the SCRIPT command or the COMMAND_FILE directive to invoke the renamed file. See the SCRIPT command for more information on command files.

WAVEFORM INPUT FILE

The Waveform Input File **plotsig.dat** is an ASCII file that serves as the input file to the program Plottime which produces waveform diagrams of the signals from the design verified. Plottime produces waveform diagrams from a Simulator output file as well as from a Timing Verifier output file. The PLOT command is used to produce the Waveform Input file. One or several Waveform Input files can be produced during a simulation session. By default the Waveform Input file is named plotsig.dat but another name or names can be specified. For more information on Plottime, see the Plottime Reference Manual.

1.8 FILE NAMES

The Simulator file names vary under different operating systems. The table below shows the file names under different operating systems for the Simulator's input and output files.

Table 1-1. Logic Simulator Input and Output Files

File	UNIX	VMS	CMS
Directives	simulate.cmd	SIMULATE.CMD	SIMULATE CMD
Expansion	cmpexp.dat	CMPEXP.DAT	CMPEXP DATA
Synonym	cmpsyn.dat	CMPSYN.DAT	CMPSYN DATA
Listing	simlst.dat	SIMLST.DAT	SIMLST DATA
Log	simlog.dat	SIMLOG.DAT	SIMLOG DATA
Waveform	plotsig.dat	PLOTSIG.DAT	PLOTSIG DATA
Command	simcmd.dat	SIMCMD.DAT	SIMCMD DATA

1.9 SPLIT-SCREEN SIMULATION

The Graphics Editor SIMULATE command creates a split-screen display (with GED in the top portion and the Simulator below) and invokes the Simulator. A sufficiently large window is required to run the split-screen Simulator; the minimum size is 48 x 86 characters.

Before simulating, the user MUST write out the drawing if any changes have been made during the current editing session and the changes are to be reflected during simulation. If Compiler errors are found, the Simulator exits and returns to the standard GED display.

Differences from Full Screen Simulation

- The user may specify a signal visible in the upper (GED) window by pointing to it with the puck instead of typing the name. For example, to open a signal, select OPEN from the menu with the puck, point to the signal to be opened, and then point to the line of the Simulator display where you want the signal to be placed (or select ; from the menu for default placement). To open an unnamed signal point to the wire.

- A command selected from the Graphics Editor window returns the user to the Graphics Editor and suspends the Simulator. A command selected from the Simulator menu, or selecting the SIMULATE command from the Graphics Editor menu, returns control to the Simulator.

- The Simulator command EXIT terminates the Simulator and causes the GED display to fill the entire window.

- If the Graphics Editor is used to change a drawing while the Simulator is running, simulation data will be inconsistent with the new version of the drawing. To simulate the modified drawing, you must EXIT the Simulator and restart simulation. It is not necessary to exit the Graphics Editor.

- "Softkeys" defined in the Graphics Editor can be used with the Simulator to save typing.

SECTION 2
LOGIC SIMULATOR DIRECTIVES

Simulator directives are parameters that control the simulation session. These directives control error reporting, I/O, and the Simulator's interpretation of the input from the Compiler. You enter directives in the Simulator directives file (simulate.cmd).

Each of the directives is described below, along with an example where usage may not be obvious. The Logic Simulator directives and their parameters are not case sensitive; each directive must be on a separate line and must be terminated by a semicolon.

BINARY_TRACE

Specifying BINARY_TRACE ON causes the Value File to be output in binary. The default, BINARY_TRACE OFF, causes the Value File to be an ASCII file. This directive is ignored for Tabular tracing.

CLOCK_INTERVALS

This directive sets the number of evenly spaced sub-periods within the clock period. For example, if there are eight sub-periods and the period of the clock is 100 ns, then MASTER CLK !C 0-2 is high from time 0 ns to time 25 ns and low from 25ns to 100ns. The directive

```
CLOCK_PERIOD 100;
```

sets the clock period to 100 ns and the directive

CLOCK_INTERVALS 20;

divides the clock into 20 intervals of equal length. With a clock period of 100 ns, each interval is 5 ns long.

For example, the signal MASTER CLK !C 0-10,15-20 is high for the first ten intervals, (that is, from 0 to 50ns) and then high again for the last five intervals, from 75 to 100 ns. The signal history for this clock is:

MASTER CLK !C 0-10,15-20 = 1:0, 0:50, 1:75

If CLOCK_INTERVALS is unspecified, the clock is divided into ten intervals.

CLOCK_ON_DRIVEN

Specifies whether clock generators may be specified on driven signals. The default for the directive is OFF, which will only permit timing assertions to be specified on undriven signals. Thus, building a clock generator on a driven signal is not allowed unless this directive is specified as ON.

CLOCK_PERIOD

Sets the period of the clock used by the Simulator. This clock period is used defining the behavior of clock signals. Clock period is specified in nanoseconds. All signals with an "!C" or "!P" timing assertion (e.g., MASTER CLK !C 0-3) have their behavior specified relative to this period. The directive

CLOCK_PERIOD 56;

sets the clock period to 56 ns. If unspecified, the Simulator sets the period to 100 ns. The clock period must be an integer and may be changed during simulation using the PERIOD command.

COMMAND_FILE

Specifies the name of a command file to be invoked as soon as the Simulator starts. This directive lets you run the Simulator in batch mode. A command file can be quickly made from the Simulator output file simcmd.dat by editing it and renaming it. The file name must be enclosed in quotes. See the SCRIPT command for a description of command files.

COMMAND_FILE '*mysetup.dat*';

COMPILER_OUTPUT

This directive is used with the SYNONYM_FILE directive to run a simulation of a design from existing Compiler files when those files do not have the default names. This directive is used to specify the name of the Compiler expansion file of the design to be simulated. When this directive is not specified, the default filename 'cmpexp.dat' is used. The file name must be enclosed in quotes. This directive is ignored when the ROOT_DRAWING directive is present.

DECAY_TIME

This directive specifies the time at which MEMORY strength signals lose their value and assume an UNDEFINED value. The default value is infinite. This means that MOS signal strengths will not decay over time unless the user explicitly specifies a decay time. The directive takes an integer. The units are nanoseconds.

DEFAULT_DRIVE

This directive is used to specify default values for rise drive and fall drive. The directive takes two values separated by commas. The first value

specifies rise drive, the second specifies fall drive. Values are given as real numbers and are used for all pins whose drive is not otherwise specified. See the section on the Delay Estimator and Expression Evaluator. Here is an example directive:

```
DEFAULT_DRIVE 0.35,0.5;
```

DELAY_ESTIMATOR

This directive is used to turn the Delay Estimator feature on and off. The default value is OFF. The directive

```
DELAY_ESTIMATOR ON;
```

enables the Delay Estimator. See the section on the Delay Estimator and Expression Evaluator.

DELAY_MODE

Delays in simulation models may be specified as a series of three values in square brackets. When this notation is used, the first value is the minimum delay, the second is the typical delay, and the third value is the maximum delay. This directive is used to select which of the three values is to be used for the current simulation run. The directive takes the values MIN, TYP, or MAX. The default value is MAX.

When using this directive, the appropriate delay values must be specified for each part used in the design. All parts with non-zero delay values must use the RISE and FALL properties or the DELAY property to specify these values using the following syntax:

```
RISE=[min,typ,max]
FALL=[min,typ,max]
DELAY=[min,typ,max],[min,typ,max]
```

If the DRIVE property is being used, minimum, typical, and maximum values must also be specified using the following syntax:

DRIVE=[min,typ,max],[min,typ,max]

Note that the square brackets are required characters in the syntax for the above properties. The values given for min, typ, and max may be either integers or real numbers. See the Delay Estimator and Expression Evaluator section.

EXP_EVALUATOR

This directive is used to turn the expression evaluator feature on and off. The default value is OFF. To enable the expression evaluator, use the directive

EXP_EVALUATOR ON;

See the Delay Estimator and Expression Evaluator section.

MEM_STATE

This directive selects between two-state memories and four-state memories. A four-state memory retains U's; a two-state memory does not. This directive takes the values 2 or 4. If not specified, memories are four-state (in fact, a misnomer since there are only three actual states). To use two-state memories use the directive

MEM_STATE 2;

OUTPUT [NO] { LIST , COMMAND_LOG } ;

This directive determines which output files are produced by the Simulator. When no directive is given, no files are created. The output file specifiers are:

LIST	causes the listing file simlst.dat to be created. The contents of the list file are controlled by other directives.
COMMAND_LOG	is a file containing all of the commands that the Simulator processed. After renaming, this file can be used as an input command file (either using the COMMAND_FILE directive or the SCRIPT command).

PIN_DELAY

This directive specifies whether pin-to-pin delays will be used by the Simulator. The values for this directive are ON and OFF. When ON is specified, the pin delay properties override the body delay properties; otherwise, the body delay properties override the pin delay properties. The default state for this directive is OFF. When pin delay values are not defined for a particular pair of input and output pins, the body delay values are assumed.

REALCHIP_LIBRARY

Specifies the name of the Realchip library file containing the full set of Realchip device definition blocks for primitives modeled by Realchip reference elements. This directive must be used if any Realchip models are used by the Simulator. Otherwise, this directive can be omitted. The file name must be enclosed in quotes. An example directive is

REALCHIP_LIBRARY '*realchip.dat*';

REMOTE_HOST

Specifies the name of the machine or machines on the network containing Realchip/Realmodel modeling hardware and the Networked Realchip Server software to serve as a remote host for simulation. This directive enables the NETWORKED mode of operation, in which patterns can be sent to remote modeling hardware over the network connection. If this directive is not specified, the HOSTED mode will be used, accessing local modeling hardware directly. Each host name must be enclosed in quotes. When several host names are given, they are separated by commas. Here is an example directive:

REMOTE_HOST '*vserver*', '*hostsim*';

RESOLUTION

This directive specifies the time resolution to be used by the Simulator. The value of the directive is a real number that specifies nanoseconds. The default value is 1. Numbers smaller than 1 are used to specify finer resolution, numbers larger than 1 are used to specify coarser resolution. The resolution currently in use by the Simulator is indicated in the display area under the label "Scale". The directive

RESOLUTION 0.05;

means that each tick on the time scale used in Waveforms mode no longer represents one nanosecond, but now represents 0.05 ns. Values specified in ns (clock period, delays, decay times, etc.) will remain in ns, but are scaled on the display (e.g., a clock period of 100 ns will appear on the display with a period of 2000 ticks; "DECAY_TIME 5000" will cause memory signals to change value after 100000 ticks). Screen-oriented commands (SIM, WAVE, HISTORY, CURSOR, etc.) maintain their relation to ticks on the screen, although the "real" times associated with those ticks has changed.

The command WAVEFORMS 0 1000 displays a time scale of 0 to 1000 ticks, but each tick now represents 0.05 ns.

Exercise caution when changing the resolution. Too fine a resolution will decrease execution speed (simulating for hundreds of ticks even when no events are scheduled) or generate massive amounts of signal histories. Before decreasing the resolution, ensure that the specification of other time values is correspondingly coarse (e.g., "RESOLUTION 50" probably will not make sense with a 20 ns clock period).

RISE_FALL

This directive specifies whether separate RISE/FALL delays will be used by the Simulator. When the RISE_FALL directive is ON, simulations are performed using both the rise and fall delays specified for parts and between pins on parts. When RISE/FALL or PRISE/PFALL values are not specified, the values of the DELAY and PDELAY properties are used. When the RISE_FALL directive is OFF, the values of the DELAY and PDELAY properties override. When the RISE_FALL directive is OFF and neither DELAY nor PDELAY properties are used, delay values are derived from the delay time if only one value is given (RISE or FALL but not both, or PRISE or PFALL but not both) or using the greater of the rise and fall delays. The default state of this directive is ON.

The following table summarizes the delay values that are used. The notation "a:b" indicates that the value "b" is assumed if the value "a" is not defined.

RISE_FALL	ON		OFF	
	RISE	FALL	RISE	FALL
no DELAY	r:0	f:0	max(r,f):0	
DELAY= x	r:x	f:x	x	x
DELAY= x,y	r:x	f:y	max(x,y)	

When the use of the separate rise/fall delay feature is specified, the part delay used for the various transitions is as follows (where X indicates any value):

OLD VALUE	NEW VALUE	DELAY USED
X	0	fall
X	1	rise
X	U	min(rise,fall)
0	Z	rise
1	Z	fall
U	Z	max(rise,fall)

ROOT_DRAWING

This directive specifies the name of the drawing on which you wish to perform simulation. Using this directive, the Simulator calls the Compiler directly, and only as needed. The name of the drawing is enclosed in quotes, like this:

```
ROOT_DRAWING 'counter';
```

To run a simulation using an existing Compiler expansion file, omit this directive.

SESSION_LOG

This directive specifies whether a copy of terminal I/O is to be output to the listing file. The directive takes the value ON or OFF. Note that SESSION_LOG ON and OUTPUT NO LIST are incompatible.

SIGNAME_CHARS

This directive defines the number of character columns dedicated to signal names on the left side of the screen in WAVEFORMS mode. The default value is 24. This directive can take the values 9 through 24 inclusive. Values outside the legal range will be rounded to the closest legal value. Here is an example directive

```
SIGNAME_CHARS 18;
```

As the number of characters is decreased, the space available for waveforms is correspondingly increased; however, with fewer characters available for signal names, a greater number of characters will be truncated when the length of the signal names exceeds the space available. Use the PEEK command to see the full signal name.

SYNONYM_FILE

This directive is used in conjunction with the COMPILER_OUTPUT directive to perform a simulation on a design from existing Compiler files when the Compiler files do not have the default names. This directive gives the name of the synonyms file. If no synonyms file is specified with this directive, the default filename 'cmpsyn.dat' is used. The file name must be enclosed in quotes. This directive is ignored when the ROOT_DRAWING directive is present.

TABULAR_TRACE

This directive specifies the trace format. TABULAR_TRACE OFF, the default, specifies standard trace format, while TABULAR_TRACE ON specifies tabular trace format.

TERMINAL

This directive specifies the terminal type. The allowed values are:

ANNARBOR
CLUSTER
GCLUSTER
GRAPHICS
VT100
3270
TTY

CLUSTER designates a SCALDsystem terminal running the Simulator locally or in transparent mode connected to the host computer.

GCLUSTER designates a SCALDsystem terminal with graphics capabilities.

GRAPHICS designates an IBM PC or a MicroVAX II.

ANNARBOR designates an Ann Arbor Ambassador terminal with 48 lines.

TTY designates any dumb video terminal or a teletype.

VT100 designates a DEC VT100 (or equivalent) with 24 lines.

3270 designates an IBM 3270 or equivalent.

If the Simulator is running in a Graphics Editor window, the TERMINAL directive is ignored.

TIMING_CHECK

This directive is used to enable timing checkers used in simulation models. The directive takes the values ON or OFF. When this directive is ON, the Simulator recognizes all the timing checker primitives used in the simulation models for the circuit and performs timing violation checking accordingly. When TIMING_CHECK is OFF, the Simulator ignores all the timing checkers and performs no timing violation checking. For designs you are validating with both ValidTIME and ValidSIM, use TIMING_CHECK OFF. The Simulator does not need to do timing analysis and this will speed simulation. The default value for this directive is OFF.

TRACE_RADIX

This directive specifies the radix to use for tabular tracing. The directive can take the values 2, 8, 10, or 16. The default value is 2. Here is an example directive.

```
TRACE_RADIX 16;
```

USE_IF

This directive is used to include two sets of directives in a single directives file: one set for batch simulation, one set for interactive simulation. The USE_IF directive takes one of two values: BATCH or INTERACTIVE. The following directives are only used if the Simulator is run in the specified mode. The USE_IF directive has effect until the next USE_IF directive, or until the end of the directives file. The following example of the USE_IF directive instructs the Simulator to use a command file, create a session log, and set the terminal type to TTY when the Simulator is run as a batch process. This directive is useful primarily on systems such as the VAX where programs can be run either interactively or as a batch process.

```
USE_IF BATCH;
COMMAND_FILE 'BATCHSIM.CMD';
TERMINAL TTY;
SESSION_LOG ON;
```

USE_REALFAST

This directive controls the use of the Realfast Simulation Accelerator and the Realmodel Modeling System. The directive takes the values ON or OFF. When enabled (USE_REALFAST ON;), a simulation is aborted if the Simulator cannot access the Realfast hardware (Realfast currently can not be shared between work stations; simultaneous use by more than one work station is prohibited). Simulation results using Realfast are the same as without its use - Realfast simply increases the speed of simulation. If this directive is omitted, Realfast is not used.

USE_SYNONYM

This directive instructs the Simulator whether or not to read the Compiler's synonyms file. Not reading the synonyms file decreases simulation loading time; however, signals can then only be referenced by their base names. The default is ON (i.e., the synonyms file is read).

USER_EXPRESSION *expr_id* (*param...*) = *equation*;

This directive is used with the Expression Evaluator. When the directive EXP_EVALUATOR OFF; is used, this directive is ignored. This directive is used to define a user-defined delay equation for computing primitive delays. *expr_id* is a name assigned by the user to the expression being defined. This is followed by a list of *params*, enclosed in parentheses, which are the variables used in *equation*. *equation* is some combination of boolean and arithmetic operations which describe an equation to be used in

computing the delay. See the Delay Estimator and Expression Evaluator section. Here is an example directive:

```
USER_EXPRESSION BufDelay(Cof)=
[load<=Cof]'drive * load', [Cof<load<=4
* Cof]'1.5 * drive * Cof';
```

USER_PARAMETER *param_id* = (*x* [, *x* ...]) ;

This directive is used with the Expression Evaluator. When the directive EXP_EVALUATOR OFF; is used, this directive is ignored. This directive is used to define user-defined parameters used in the delay equation specified with the USER_EXPRESSION directive. *x* is a real number. Up to five real numbers may be specified and should be enclosed in parentheses. *param_id* is a name assigned by the user to the set of parameters being defined. See the USER_EXPRESSION directive and the Delay Estimator and Expression Evaluator section. Here is an example directive:

```
USER_PARAMETER INV1=18;
```

WIRE_DELAYS

This directive specifies the name of the wire delays file; the filename must be quoted. See the Delays section. Here is an example directive:

```
WIRE_DELAYS 'wiredel.dat';
```

WIRE_ESTIMATE

This directive is used with the Delay Estimator. It allows the user to define a look-up table specifying equivalent loads of a net. The look-up table contains a list of values corresponding to the delays associated with incrementally increasing load. Up to 100 real values may be specified. The values are separated by commas, like this:

WIRE_ESTIMATE 5.5,6,6.4,6.8,7.1,7.5;

The directive can also take a family name parameter, so that different look-up tables can be used for different families of parts. Here is an example:

WIRE_ESTIMATE TTL 5.5,6,6.4,6.8,7.1,7.5;

The body property FAMILY is attached to all bodies for which the alternate look-up tables are to be used. See the Delay Estimator and Expression Evaluator section for more information.

SECTION 3
WAVEFORMS MODE

This section describes how to use the Simulator in Waveforms mode. Waveforms mode displays the value of each signal as a waveform over time. Using Waveforms mode, you can see twelve signals at one time in the split screen Simulator under GED, or up to 48 signals at one time in the full screen Simulator.

A certain group of commands that are not appropriate for use in Bus mode, are used in Waveforms mode. These commands are:

WAVEFORMS	ROW
HISTORY	SCROLL
CURSOR	SPACING
DELTA_TIME	

Each of these commands are described below. In addition, the OPEN command is also described below because it works differently under Waveform Mode than it does under bus mode.

Many of these commands take a number argument that designates time in nanoseconds. These number arguments are all affected by the value of the RESOLUTION directive. When RESOLUTION = 1 (the default) each tick mark on the Waveforms display represents 1 ns and the number arguments represent nanoseconds. When RESOLUTION = 10, for example, each tick mark represents 10 ns and the number arguments represent tick marks. The command

```
CURSOR 100
```

moves the Cursor to the tick mark labeled 100, which actually represents 1000 ns.

3.1 WAVEFORM COMMANDS

The following commands are commonly used in the Waveforms mode and affect the signal display.

WAVEFORMS *start_time end_time*
or
WAVEFORMS *pt1 pt2*

This command has two separate functions: the first is to invoke Waveforms mode and define the range of time to be displayed; the second is to pan and zoom on displayed waveforms to get a better view. Because Bus mode is the default, the first time you give the WAVEFORMS command, it invokes Waveforms mode. Be sure that you give both a start time and an end time. The Waveforms display advances automatically with simulation.

While you are in Waveforms mode, the WAVEFORMS command is used to pan and zoom the display. Several different syntaxes are available for using the waveforms command to pan and zoom.

1. The most common form of the WAVEFORMS command is

   WAVEFORM 100 300

   The two numbers designate real time in nanoseconds. The first number is the start time and the second is the end time. On systems with a puck or mouse, the times can be designated as points on the waveforms display (to zoom in) or as points on the time line in the echo area (to zoom out). If the *end_time* or *pt2* is omitted, the current display width (*end_time* - *start_time*) is used with the new *start_time*. The screen pans to the right.

2. The times can also be given as *relative* times, by using this syntax:

   WAVEFORM left 100 right 400
   WAVEFORM right 100 right 50

In this syntax the start time (the first parameter) is relative to the existing start time, and the end time (the second parameter) is relative to the new start time.

HISTORY *time*

The History command is used to set the time period during which signal history for each opened signal is maintained. When Waveforms mode is entered, HISTORY is automatically set to 10000 for all OPENED signals. Setting HISTORY to a smaller number can improve performance. When HISTORY is 500, signal history is only kept for each OPEN signal for the 500 ns prior to the current time. The HISTORY command is affected by the RESOLUTION directive. When using a large number for resolution, you will probably want to set HISTORY to a smaller number.

The HISTORY command with no argument tells you the current value for HISTORY.

CURSOR *time*

This command moves the cursor to a new time. The *time* parameter may be specified by giving a number, or using a puck press on the waveforms display or on the time line in the echo area. When a number is given, the units are nanoseconds, and are affected by the value of the RESOLUTION directive.

A relative time can be specified by using the syntax:

```
CURSOR LEFT 25
CURSOR RIGHT 100
```

Whenever the cursor is moved, the signal values on the right side of the screen are changed to indicate the signal values at the cursor time. The cursor may be set to any time between 0 and the current time, whether the new time is visible or not. In full-screen graphics mode, if the new time is visible, a vertical line is drawn through the entire waveform display area; this line is displayed on all empty

rows including the blank rows between waveforms when SPACING is greater than 1. When simulation is complete, the cursor is automatically moved to the current time.

In full-screen graphics mode a timeline appears in the echo area and the value for the CURSOR command can be given by a puck press on this line. The timeline does not appear when running split-screen simulation under GED.

DELTA_TIME *pt1 pt2*

The DELta_time command is used to determine the time difference between two points on the current waveform display. The points are specified using the puck, so this command is only available on systems having a puck. The value appears in the echo area. Points can be specified anywhere in the current waveforms display.

OPEN *signal* [, *row* [, *col*]]
or
OPEN *signal pt* [(*dest pt signal pt*)...] [*dest pt*]

The OPEN command adds a signal to the display, and in Waveforms mode it starts recording signal history for that signal.

In split-screen simulation, a signal can be opened by selecting it with the puck in the GED drawing, and either giving a destination point with the puck or ending the command.

When a destination point is given the signal is opened on that line of the display. When none is given, the signal is opened on the first available line (if the signal has not been opened before). You can open a maximum of 200 signals. The command must be terminated by a semicolon or carriage return.

The user can replace an existing signal by opening a new signal and specifying *row*. Once a signal is OPENed in Waveforms mode, the history for the signal is maintained for the specified history period, even if the signal is not on the screen. To remove an OPEN signal, use the REMOVE command or OPEN another signal on the same row. This feature allows a user to OPEN more signals than can be displayed at once, simulate to calculate their signal behavior, and then view their behavior.

ROW *number*

This command controls which signals are displayed on the screen. The *number* gives the row number for the row that is to be placed at the top of the screen. Relative row numbers may also be specified by preceding the number with a + or - sign, or by pointing to the row using the puck. The number of the top row currently displayed on the screen is shown as one of the fields on the status line. Note that when changing the top row, all signals previously displayed above the new top row are scrolled up and off of the screen.

SCROLL

This command allows the user to control the automatic scrolling feature of the Simulator. The command takes the argument ON or OFF. In Waveforms mode, when you have a full-screen of signals, the Simulator normally scrolls the display to OPEN each additional new signal. If you have many signals to open, it is faster to turn SCROLL off, until OPEN and DEPOSIT is completed. Using SCROLL OFF allows the user to OPEN and DEPOSIT into signals that are not on the display.

SPACING *value*

This command allows the user to specify single or double (or multiple) spacing between adjacent waveforms in Waveforms mode. *value* is an integer indicating the number of spaces between waveforms. The default is 1

(single spacing). The current SPACING value is displayed in the echo area. Whenever the value of SPACING is changed, the screen is redrawn using the new value of SPACING.

SECTION 4
BREAKPOINTS AND PATCHING

Breakpoints are triggering conditions that cause the Simulator to stop simulating and accept commands from the user. The following are some important uses of breakpoints:

- Skipping to a point of interest; for example, when a shift register shifts to all zeros.

- Performing "background" tests while the user stimulates the design (such as stopping whenever the design enters an error condition).

The logic patching facility allows the user to make simple modifications to a design without recompiling. This facility is useful primarily for "tacking" bug fixes in before they are entered into the design, or for stimulating an incomplete design. Some example uses are:

- Patching a design by forcing signals to some state, such as forcing the PARITY ERROR signal to a 0 whenever some pattern is read that is incorrectly reported as an error.

- Generating test stimuli based on the state of the design, such as submitting instruction N+1 whenever instruction N has completed.

The Patch facility allows you to temporarily assign a particular signal value to a signal during a simulation session. The commands you use are similar to the breakpoint commands.

4.1 BREAKPOINTS

Breakpointing conditions are boolean expressions of signals present in the design (refer to expression syntax). A breakpoint is encountered (triggers) when the expression defining it changes value from false to true.

In addition to the standard boolean operators AND, OR, XOR, and NOT, state information can be included in breakpoint expressions to allow the user to build a state machine that detects when to trigger a breakpoint. This general form of the trigger-enabling feature is used in most logic analyzers.

To simplify construction of complex breakpoints, a new class of signal called an ENABLE signal has been added to the Simulator. ENABLE signals never exist in a design, but are created by the user as partial products in expressions. ENABLE signal names follow the same rules as standard signal names, but they are always scalars.

Simulation halts to display a breakpoint expression, but the system remains in interactive mode. The user may perform other operations or may continue simulation using another SIMULATE command.

If a breakpoint is encountered during execution of a SIMULATE command in a command file, simulation time stops at the breakpoint and the next command in the script is executed. This allows the user to create scripts for circuits where it is unknown how long to simulate before a certain event will occur.

Note that breakpoints should not be used when operating REALFAST.

4.2 BREAKPOINT COMMANDS

This section contains a list of commands used with breakpoints. See the following section for a description of breakpoint syntax.

SET ENABLE *signal* WHEN *expr*

Sets *signal* to 1 when *expr* is true. The signal is a "new" signal that is created the first time it is referenced by the user (i.e., the signal cannot already exist in the design).

CLEAR ENABLE *signal* WHEN *expr*

Clears *signal* to 0 when *expr* is true. The signal is a "new" signal that is created the first time it is referenced by the user (i.e., the signal cannot already exist in the design).

SAMPLE ENABLE *signal* GETS *expr 1* WHEN *expr 2*

Equates *signal* to the value of *expr 1* when *expr 2* changes from a 0 to a 1. The signal is a "new" signal that is created the first time it is referenced and cannot already exist in the design.

LATCH ENABLE *signal* GETS *expression 1* WHEN *expr 2*

Equates *signal* to the value of *expr 1* when *expr 2* is a 1. The signal is a "new" signal that is created the first time it is referenced and cannot already exist in the design.

EQUATE ENABLE *signal* TO *expr*

Continuously gives *signal* the value of *expr*. The signal is a "new" signal that is created the first time it is referenced and cannot already exist in the design.

SET BREAKPOINT *expr*

Installs *expr* as a breakpoint. While this breakpoint is set, the Simulator ALWAYS stops when the function changes from false to true. When the simulator stops, it prints out the function to identify which breakpoint was encountered. The Simulator assigns numbers to breakpoints to allow a breakpoint to be specified either by number or function; simple breakpoints can be called by name, and complex breakpoints can be called by number.

Named breakpoints are a special case of ENABLE signals. A user can EQUATE an ENABLE signal to the desired breakpointing expression, and thereafter reference the breakpoint by the name of the ENABLE signal as follows:

EQUATE ENABLE *name* to *breakpoint_condition*
SET BREAKPOINT *name*

SET BREAKPOINT # *number*

Activates the indicated breakpoint. While this breakpoint is set, the Simulator ALWAYS stops when the function changes from false to true. When the simulator stops, it prints out the function to identify the responsible breakpoint. Breakpoints are given numbers by the Simulator, and complex breakpoints may be re-installed by number. (Note that when simulating under CMS, the # symbol prefix must be replaced by the % symbol).

CLEAR BREAKPOINT *name*
or
CLEAR BREAKPOINT # *number*

Deactivates the breakpoint. The breakpoint no longer affects simulation but it remains in the breakpoint list marked "inactive". Only breakpoints that are a single string containing no boolean expressions may be cleared by name; all others must be cleared by number. Breakpoints you can clear by name are usually those defined using the EQUATE ENABLE command.

LIST BREAKPOINTS

Lists all breakpoints that have been created, whether they are active or have been CLEARed. Breakpoints are marked as active (SET) or inactive (CLEARed). This command also prints the breakpoint number assigned by the Simulator.

LIST ENABLES

Lists all of the ENABLE signals that have been defined and their definitions.

4.3 COMBINING BREAKPOINT COMMANDS

Groups of SAMPLE, LATCH, SET, CLEAR, and EQUATE commands may be applied to any signal, with the following results:

- EQUATEing a signal generates a combinational function only; signals generated with the EQUATE command have no state of their own. EQUATEing a signal that has already been equated supersedes the old definition.

- SAMPLEing, LATCHing, SETting or CLEARing a signal generates a function containing state information. SETs and CLEARs may be added to a signal that is SAMPLEd or LATCHed. A SAMPLE or LATCH may be added to a signal that is SET and/or CLEARed. Defining a SAMPLE, LATCH, CLEAR, or SET for a signal that already has such a definition supersedes the old definition. Only one SAMPLE or LATCH definition applies at one time. Defining a SAMPLE or LATCH for a signal already defined to have the other definition supersedes the existing definition.

- Only one EQUATE definition or one definition from the set {SAMPLE, LATCH, SET, CLEAR} applies at a time. EQUATEing a signal that was previously defined as SAMPLEd, LATCHed, SET, or CLEARed supersedes the existing definition. SAMPLEing, LATCHing, SETting, or CLEARing a signal that was previously EQUATEd supersedes the existing definition.

4.4 EXPRESSION SYNTAX

The syntax for an *expression* is based on the SCALD standard expression syntax:

```
<expression> -> <expression> OR <boolean expression>  { boolean
        -> <expression> XOR <boolean expression> { boolean XOR
        -> <boolean expression>

<boolean expression>   -> <boolean expression> AND
                    <relational expression>  { boolean AND }
                -> <relational expression>

<relational expression> -> <term> <rel OP> <term>
                -> <term>

<rel OP>    -> <'='>            { equal }
        -> <'<>'>           { not equal }
        -> <'>='>           { greater than or equal }
        -> <'<='>           { less than or equal }
        -> <'<'>            { less than }
        -> <'>'>            { greater than }

<term>      -> <signal>
        -> ( <expression> )
        -> NOT <term>       { boolean NOT}
        -> 0            { constant 0 }
        -> 1            { constant 1 }
        -> & <constant>     { any constant, given
                        in current radix }
```

A *signal* can be any number of bits wide, but the *expression* used in a breakpoint or enable definition must evaluate to a single bit. If a vector signal is used without a subscript, the entire width is considered for the expression. The AND operator takes precedence over the OR operator and the XOR operator. A *rel OP* takes precedence over any boolean operator except the NOT operator; when a *rel OP* is used, it should be separated from *terms* by spaces to prevent confusion in parsing. Some useful examples are:

```
SET BREAKPOINT NOT OUTA
SET BREAKPOINT DATA<15..0> >= &3F0
SET BREAKPOINT ( read* = 0 OR write AND
refresh ) = 0
```

The last expression is evaluated as follows:

SET BREAKPOINT (((read* = 0) OR (write AND refresh)) = 0)

4.5 LOGIC PATCHING

The logic patching facility allows the user to redefine the behavior of a scalar signal or a single bit of a vector signal in the design by specifying the new behavior of the signal as a boolean expression of signals in the design (refer to the expression syntax of signals). Note that patching must be done after the LOGIC_INIT command. The commands and operators used to patch a signal are very similar to those used for defining breakpoints:

SET PATCH *signal* WHEN *expr*

Sets *signal* to 1 when *expr* is true. The signal must be present in the design.

CLEAR PATCH *signal* WHEN *expr*

Clears *signal* to 0 when *function* is true. The signal must be present in the design.

SAMPLE PATCH *signal* GETS *expr 1* WHEN *expr 2*

Equates *signal* to the value of *expr 1* when *expr 2* changes from a 0 to a 1. The signal must be present in the design.

LATCH PATCH *signal* GETS *expr 1* WHEN *expr 2*

Equates *signal* to the value of *expr 1* when *expr 2* is a 1. The signal must be present in the design.

EQUATE PATCH *signal* TO *expr*

Continuously gives *signal* the value of *expr*. The signal must be present in the design.

LIST PATCHES

Lists all PATCH signals that have been defined and their definitions.

SECTION 5
TRACING AND TABULAR I/O

Tracing is a way to output the state of the design at various times during the simulation. This type of report can be generated during batch simulation and examined interactively. Two formats for trace generation are described here: the standard trace format and the tabular trace format. The tabular form may also be used as input to the Simulator to force signals to values or patterns at specified times (see the section below, "Stimulating Circuits with Tabular I/O files"). See the TABULAR_TRACE Simulator directive for information on how to specify which trace format to use.

5.1 TRACING

REQUIREMENTS FOR STANDARD TRACING

A program reading the trace must be able to find the value of any signal at any time during the simulation. The required information may be separated into CONNECTIVITY information, which describes the circuit, and VALUE information, which describes the state of the design. The connectivity of a design is nearly constant during a simulation; it is modified only by explicit logic patching or breakpoint generation commands from the user. The portion of connectivity that is most useful for understanding the behavior of a circuit is the mapping between outputs and signals. The value information of a design changes very rapidly during a simulation and includes all state transitions occurring in the design. The Simulator trace output is placed in two files: the signal mapping file and the value file.

SIGNAL MAPPING

A program reading trace output needs a description of how signals are attached to outputs in order to relate simulation results to the circuit. An output of a part connects to a range of bits of a signal or signals. The signal mapping file relates which bits of which signals attach to which bits of which outputs.

VALUE INFORMATION FOR TRACING

Value information is output in both absolute and relative form. At the beginning of the simulation, and possibly at intervals throughout the simulation, the state of the entire design is output in absolute form. As each output pin changes state, that change is reported in relative form. A program may extract some or all transitions in the design by reading just the relative sections of the value file, or may maintain the current state of the design by first reading an absolute report, and then applying transitions as they appear in the relative reports.

FILE FORMATS FOR TRACING

Both the signal mapping file and the value file contain *output descriptors* and *primitive segment descriptors.* Each of these is a unique 32-bit integer that represents an output or primitive segment. In the signal mapping file and the ASCII version of the value file, these descriptors are output as signed decimal integers. In the binary version of the value file, they are output as binary integers. Any 32-bit integer may be used as either an *output descriptor* or a *primitive segment descriptor* but not both. Therefore, it is possible to determine the type of a descriptor from its value.

Several *output descriptors* are reserved for use as sentinels in the value file. A sentinel is a reserved value that has a special meaning, such as the last element in a list. Any sentinel *output descriptor* will not be used as either a true *output descriptor* or a *primitive segment descriptor.* The values of these sentinels may differ from simulation to simulation, and are defined in the signal mapping file.

SIGNAL MAPPING FILE FORMAT

The signal mapping file has the following format:

```
STATE_ENCODING
  <list of state encodings>
RELATIVE_SENTINEL <relative sentinel descriptor> ;
ABSOLUTE_SENTINEL
  <absolute sentinel descriptor> ;
END_FILE_SENTINEL <end file sentinel descriptor> ;
RESERVED
  <reserved information>
END_RESERVED ;
SIGNAL_MAPPING
  <list of signal mappings>
END_SIGNAL ;
MEMORY_MAPPING
  <list of memory mappings>
END_MEMORY ;
```

list of state encodings is a list of the following entries:

```
<state name> : <state value> ;
```

state name is the name of the state in single quotes. *state value* is a 32-bit integer that describes the value representing the state.

<relative sentinel descriptor> is the special <output descriptor> that indicates that a relative report follows. <absolute sentinel descriptor> is the special <output descriptor> that indicates that an absolute report follows. <end file sentinel descriptor> is the special <output descriptor> that indicates that the end of the file has been reached.

<reserved information> is a portion of the file that has not yet been defined, except that it is terminated by the keyword END_RESERVED.

<list of signal mappings> is a list of the following entries:

```
<signal name> <subrange> = <output descriptor> :
        <offset>, <is bubbled> ;
```

<signal name> is a single-quoted string that contains a base signal name in canonical syntax. <subrange> is a subrange of the signal of the form:

"<" <most significant bit> .. <least significant bit> ">"

or

"<" <single bit number> ">"

or

<nothing>

If no *subrange* is specified, the signal is a scalar. *output descriptor* identifies the output to which the signal is connected. *offset* is the bit number on the specified output that matches the least-significant bit of the signal subrange. *is bubbled* is BUBBLED if the output pin is bubbled, and is NOT_BUBBLED if the output pin is not bubbled.

list of memory mappings is a list of the following entries:

<memory path name> <subrange> =
  <primitive segment descriptor>, <is bubbled> ;

memory path name is the path name of a memory in the design and is enclosed in single quotes. *subrange* describes a contiguous subrange of the memory, that matches the *primitive segment descriptor.*

primitive segment descriptor describes which primitive segment matches the indicated subrange of the memory. The bits of a memory are numbered in increasing bit numbers from 0, which is least significant, to *size*-1, which is most significant. *is bubbled* is BUBBLED if the memory output pin is bubbled, and is NOT_BUBBLED if the memory output pin is not bubbled.

VALUE FILE FORMAT

The value file contains a list of the following entries:

<type sentinel> <absolute time> <list of output states>

<type sentinel> equals an ABSOLUTE_SENTINEL, a RELATIVE_SENTINEL, or an END_FILE_SENTINEL. <absolute time> is a 32-bit integer. <list of output states> lists output pins and their current states. Each entry in the <list of output states> has the following format:

<output descriptor> <value 1> <value 2>

or

<primitive segment descriptor> <memory address> <value 1> <value 2>

<output descriptor> describes an output pin. <value 1> and <value 2> are 32-bit integers that represent the state of the output pin. The states of the eight bits of the output pin are represented in the eight bytes of <value 1> and <value 2>. The highest order output pin bit is in the highest order byte (bits 31..24) of <value 1>. Lower order output pin bits are stored in descending bytes ending with the lowest order output pin bit in bits 7..0 of <value 2>. This list continues until another <type sentinel> is reached. If the <type sentinel> equals an ABSOLUTE_SENTINEL, then the following state information represents an absolute report. If the <type sentinel> equals a RELATIVE_SENTINEL, then the following state information represents a relative report. If the <type sentinel> equals an END_FILE_SENTINEL, then this is the last entry in the file, and no <absolute time> or <list of output states> follows.

primitive segment descriptor describes a memory primitive segment. *memory address* is a 32-bit integer that indicates which location in the memory has changed. *value 1* and *value 2* are 32-bit integers that represent the new state of

the memory location.

Whether the first element is an *output descriptor* or a *primitive segment descriptor* can be determined by checking which way the integer was referenced in the signal mapping file.

This file format is optimized for binary representation, but is supported as both a binary and as an ASCII file of signed decimal integers.

5.2 TABULAR I/O

FILE FORMAT FOR TABULAR I/0

The Tabular I/O output file includes a list of the signals being traced, the radix in which they are being traced, and a series of records that specify times and signal values; signal strengths are not output. Values that have not changed since the last time specification need not be specified again, although a comma is still required to delineate the field. The following is an example of a Tabular I/O file:

```
FILE_TYPE = TABULAR_TRACE;
sig1,2
sig2<10..8>,8
sig3<5>,2
sig2<7..0>,2
START_TAB_TRACE;
        0 / U,U,U,UUUUUUUU;
       10 / 1,1,Z,10010110;
       20 / 0,5,U,;
       30 /  ,,U,10010111;
 .
 .
 .
END_TAB_TRACE;
END.
```

This example shows four subranges of three signals being traced every 10 nanoseconds. Signals can be traced at any interval or at every transition; that is, a new record is produced if a change occurs in any one of the signals being

traced. See the TRACE_INTERVAL command for more information.

STIMULATING CIRCUITS WITH TABULAR I/O FILES

The Simulator can read in a Tabular I/O file, such as the one in the example above, and set the signals specified in it to the specified values at the specified times. The Simulator reads the time from the file, and when the time in the simulation reaches this value, the signal values are read and deposited into the signals specified in the first part of the file. See the TRACE_READ and TRACE_RESET commands for further details.

The file can be created "manually" with a text editor or by the Simulator from a previous run. When creating a Tabular I/O file manually, note that signal names containing a comma should be enclosed in quotes (e.g., 'CLK !C0-1, 3-4'). Also note that if the values, which are separated by commas, extend beyond an 80 character line, a tilde ~ must be entered at the end of the line if a value is to be continued on the next line; that is, values cannot appear in the 80th column (although a "," or ";" is acceptable). If the values extend over 255 characters, put a new line character before the signal value that would make the total number of characters exceed 255. For example:

```
10 / 1,1,0010101, ... 010,UU11011011,UUUUUU~
UUUU,Z101011Z,10, ... 101,UUUUU,1,11,1,1111,
1,11,0101011,1,1, ... 001,1,1,1,1,1,1,1,10,~
010100,
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ,1,11;
25 / 0,1,0010101, ... 011,      ,0000000000, etc.
```

In this case, a new line was inserted before the Zs because, otherwise, the 255 character limit would have been reached. Signals wider than approximately 250 bits must be split into multiple segments to be traced using Tabular I/O. Note that memories cannot be traced using Tabular I/O.

SAMPLE TABULAR I/O USE

With the directive "TABULAR_TRACE ON;" included in your directives file, the following command sequence might be given to create a tabular stimulus file:

```
trace_radix 16      { trace buses in hex }
trace input         { specify signals to trace }
trace output
trace sum<15..3>
trace_start         { open the file, start tracing }
sim 100             { OPEN signals,    }
dep 42
trace_stop          { stop the trace }
trace_close         { close the file; write to disk }
```

To then use the file as stimulus to the Simulator, you could use the following commands:

```
logic_init U                { set time back to zero }
trace_read tabfile.dat { read the stimulus file }
sim 800             { simulation with stimulus of }
```

If stimulus from more than one tabular input file is desired, the TRACE_RESET command can be used to reset the Simulator after each file has been used. For example, after the above sequence of commands, the user might specify the following sequence to use a different tabular input file:

```
logic_init -*
trace_reset
trace_read tabfile2
sim c
```

SECTION 6
LOADING MEMORIES

Memories are loaded from the memory contents file using the MEMLOAD command. First locate the memory using the MEMPATH command.

The format of the memory contents file is identical to the format of the file generated by the DUMPMEMORY command. A memory contents file containing four 36-bit words might appear as:

```
FILE_TYPE = MEMORY_CONTENTS;
BIT_RANGE = 35 .. 0;
MEM_BLOCK 0,4;
  0000 0001 0100 0000 1111 1010 1011 0101 1111 ;
  0000 0010 0100 0000 1111 1110 1011 1101 1111 ;
  0000 0011 0011 0000 0000 1111 1111 0110 0100 ;
  0000 0000 0101 0000 0000 1111 1111 1111 1101 ;
END_MEM_BLOCK;
END.
```

BIT_RANGE determines the word size and bit numbering of the data words in the file. Regardless of library format, the syntax for BIT_RANGE is "*high value low value*" (e.g., "35..0", not "0..35"). MEM_BLOCK is followed by two decimal parameters. The first parameter is the starting address of the block, the second parameter is the number of words in the block. Following MEM_BLOCK are the data words in binary. Each data word must be the length specified by BIT_RANGE and must end with a semicolon. Spaces may be inserted in the data words for clarity. There may be any number of MEM_BLOCKS; however, all MEM_BLOCKS must be placed in ascending order of address and must not specify overlapping ranges.

The format of the MEMLOAD command is:

MEMLoad *filename* ,< *file bit range*> , [*file word range*] , < *primitive bit range* > , [*primitive word range*]

If no filename is given, the user is prompted for one.

The four optional arguments to the MEMLOAD command are used to specify a mapping from the memory contents file to the memory primitive. Note that bit- and word-range arguments must be given as integers (in decimal) and must be enclosed in the appropriate brackets (either pointed or square, as indicated). An example of the complete command syntax is:

MEMLOAD
ramvals.dat,<8..5>,[200..100:10],<3..0>,[20..0:2]

The file word range [high addr .. low addr : step] specifies which words from the memory file are to be deposited in the memory primitive, and the primitive word range [high addr .. low addr : step] specifies the mapping of the words within the memory primitive. Thus, the example above maps words 200,190,180,170,..._. of the file into words 20,18,16,14,..._. of the memory primitive. If word ranges are not specified, they default to [*size*-1..0] where *size* is the depth of the memory primitive.

The file bit range *n* .. *m* specifies which of the file word bits are deposited in the memory primitive, and the primitive bit range specifies the mapping of the file word bits within the memory primitive. Thus, the example above maps bit 8 of file word into bit 3 of the primitive, bit 7 of the file word into bit 2 of the primitive, and so on. If bit ranges are not specified, they default to the range of the memory primitive (either <*width*-1..0> or <0..*width*-1>, depending on the signal syntax being used at the site).

SECTION 7
SIMULATOR COMMANDS

All Simulator commands take the form of a command name followed by arguments, if necessary. You may abbreviate commands if the abbreviations are not ambiguous. In the descriptions below the shortest unambiguous abbreviations are given in boldface type. All commands are terminated by either pressing RETURN or selecting the ; (semicolon) from the Simulator menu. All command inputs may be typed in either upper or lower case. Some commands prompt for arguments if none are given.

Certain commands allow puck points as arguments. On systems where a puck is not available, all arguments must be typed in. Each command is described below. The listing is alphabetical.

ASSERTIONS *signal, time specifier*

The ASSERTIONS command allows timing assertions to be specified while running the Simulator. This allows the user to specify assertions interactively rather than as part of the signal name on the GED drawing. This feature provides greater flexibility during simulations because timing assertions can be changed during the simulation run.

The *time specifier* parameter is specified using the standard SCALD syntax for timing assertions (e.g., 0-4). Do not include the assertion character ! (exclamation point), or assertion prefix (C). The Simulator automatically adds the "!C" prefix to the time specifier.

This command can be invoked on existing clock signals as well as any other signals in the drawing. Thus, any signal can be assigned a timing assertion

during a simulation run, and assertions of existing clock signals can be re-defined. After assigning clock assertions, the signal can be OPENed using either its previous or its new (with assertions) name.

BUS

This command is used to enter Bus mode simulation and refreshes the screen. In Bus mode instantaneous signal values are displayed on the screen instead of waveforms. Bus mode is effective for large designs where many signals need to be opened at a time. Bus mode simulation is the default. Waveforms mode is the other simulation mode. In Waveforms mode signals appear one to a line and their signal history is shown as a waveform. To enter Waveforms mode use the WAVEFORMS command. Signals opened in one mode are not automatically opened in the other mode. Be sure to select the appropriate mode before opening signals.

CLEAR BREAKPOINT *signal*
or
CLEAR BREAKPOINT # *number*

This command clears a breakpoint. The argument is either a signal name or # followed by a number of a current breakpoint. See the Breakpoints section.

CLEAR ENABLE *signal* WHEN *expression*

This command Clears an ENABLE signal when *expression* is true. See the Breakpoints section.

CLEAR PATCH *signal* WHEN *expression*

This command Clears a PATCH signal when *expression* is true. See the Logic Patching section.

CLOCK

This command turns the clocks on and off. CLOCK ON turns on the clocks; CLOCK OFF turns off the clocks. With no argument, CLOCK reports the ON/OFF state of the clocks. When clocks are turned off, the clock generator primitives are disabled and the values of clock signals stop changing. When clocks are turned on, at the start of the next simulation all primitives are re-evaluated, and all clock values are immediately set to their correct instantaneous values.

COMPARE *value*

This command Compares the value of the currently open signal against *value*

An indication is given of whether or not the comparison was successful. If the comparison is not successful (and the command originated from a command file), the Logic Simulator PAUSEs from the command file and returns command control to the terminal. Control may be returned to the command file with the RESUME command.

COVERAGE [ON |OFF]

Enables simple coverage analysis allowing the user to obtain a list of the signals that have made a transition during a period of simulation. With no argument, the current status of the coverage analysis is reported. If coverage analysis is off, the Simulator does not track the number of transitions. See also the WRITE_COVERAGE command.

CURSOR *time*

Moves the WAVEFORMS cursor to a new time. See the Waveforms section.

DELTA_TIME *pt1 pt2*

Indicates the time difference between two points on the current waveform display. The points are specified using the puck, so this command is only available on systems having a puck. See the Waveforms section.

DEPOSIT [*signal,* |*pt*] *value1*
[{@ |+} *time1* [*value* {@ |+} *time*]...]

Deposits *value1* to the indicated *signal* at *time1* and schedules subsequent *values* to be deposited at the indicated *times*. Multiple bit values appear in the current radix; if the number of bits in the value exceed the width of the signal, the extra bits at the high order end are ignored. The signal is an optional parameter which may be specified using the puck. *time1* and subsequent value/time pairs are also optional and may be either absolute or relative to the current time. Absolute times must be preceded by the @ character, while relative times must be preceded by the + character. In the absence of these optional parameters, *value1* is be deposited into the currently OPEN signal at the current time.

Only a space is required to separate value/time pairs. These pairs do not have to be in any specific time order. If time x = time y and value x does not equal value y, then the latest value specified is deposited to the signal. This applies to both single and multiple invocations of the command. Also, if any time x < current time, the corresponding value is ignored. Values from the DEPOSIT command always override the values specified in a tabular input file for the same time. Unlike Tabular I/O, times and values specified using the DEPOSIT command are volatile; i.e., after they are output, the Simulator retains no knowledge of them. In addition, if the simulation time is reset to 0 (using LOGIC_INIT), any scheduled deposits are cleared.

Note that this command will neither OPEN the specified signal nor change which signal is currently OPEN. If the specified *signal* has not been opened, DEPOSIT causes the value to be placed on the signal, but will neither OPEN it nor cause signal history to be started.

Neither of the optional DEPOSIT parameters may be specified when dealing with a memory location; memory locations continue to require individual DEPOSIT commands in order to change their contents.

Here is an example. If the current time is 50, the command:

```
dep input,a@ 100 b@ 200 c+ 500 d@ 30 e+ 400
f@ 100
```

deposits to the signal named INPUT the following values:

f at time 100
b at time 200
e at time 450
c at time 550

The value a does not appear because it is overwritten by the value f. The value d is ignored because 30 is previous to the current time 50.

DISPLAY [ON |OFF]

Allow the user to enable/disable updating of the display area of the screen. This is particularly helpful in increasing the Simulator's speed when continuous updating of the display area is not required. When updating is disabled, a field on the status line indicates this to the user. The output to the echo area in response to the commands given proceeds as usual. When the display is reenabled, the screen is redrawn as if a REDISPLAY command was issued.

DUMPMEMORY *filename* [,*primitive bit range*, [*primitive word range*]]

Dumps the contents of memory primitive into *filename* (the user is prompted if a filename is not specified). The optional bit and word range parameters specify a window of memory to be dumped; note that both ranges are taken to be in decimal regardless of the current radix. If no optional parameters are specified, the entire memory is dumped. The file created can be used to load the memory with the MEMLoad command. See the Loading Memories section.

EQUATE ENABLE *signal* TO *expression*

Equates an ENABLE signal to *expression*. See the Breakpoints section.

EQUATE PATCH *signal* TO *expression*

Equates a PATCH signal to *expression*. See the Logic Patching section.

ERASE

Erases the entire display area of the screen, including all signals and values. Resets the top row number to 1 and restores the status lines.

EXIT

Exits the Logic Simulator.

HARDCOPY [{ A - E }]

Produces a plot of the current Simulator screen. This command only works when the Simulator is running under GED or with the graphics Simulator. When running under GED, the plot is produced by

GED and the optional parameter is not available. The parameter may be specified with the graphics Simulator to produce an output of the desired page size; the default is "A".

Several plotter types are supported and a local/spooled option is available through the SET command (see below).

HISTORY *recording period*

Sets or provides the recording period for WAVEFORMS. See the Waveforms section.

INIT_COVERAGE

Clears the list of signals that have made a transition. This command enables the user to invoke coverage analysis for different periods of simulation (see the COVERAGE command). Note that turning coverage analysis OFF does not clear this list - this command must be invoked each time a new list of signals is to be started (except the first, when the list is empty), regardless of the use of the COVERAGE command.

INTERVAL *value*

Sets the number of clock intervals to the specified decimal integer *value*. The interval value appears on the status line. If the number of intervals is too small, a warning is given.

LATCH ENABLE *signal* GETS *expression 1* WHEN *expression 2*

Latches an ENABLE signal to *expression 1* when *expression 2* is true. See the Breakpoints section.

LATCH PATCH *signal* GETS *expression 1* WHEN *expression 2*

Latches a PATCH signal to *expression 1* when *expression 2* is true. See the Logic Patching section.

LIST BREAKPOINTS

Lists all breakpoints. See the Breakpoints section.

LIST DEPOSITS

Lists all signals with associated value/time pair(s) for which DEPOSITs have been scheduled at a time > current time. The times reported are absolute times.

LIST ENABLES

Lists all ENABLE signals. See the Breakpoints section.

LIST PATCHES

Lists all PATCH signals. See the Logic Patching section.

LIST SIGNALS

Lists all signals originally present in the design. Breakpoints and patches applied to the signal also are reported.

LIST TRACES

Lists all signals, subranges, and memories that are currently being traced along with the radix in which they are being traced. For example, if List Traces is typed after the three trace commands in the example shown in the Trace command section, below, the

following output appears:

```
data,binary
out<66..33> ,hex
midout<4> ,binary
```

LOADMEMORY

This is another name for the MEMLOAD command.

LOGIC_INIT { 0 |1 |* |-* |U }

Resets simulated time to 0 and initializes all signals to the specified value. Note that this command does not alter the contents of memories. "*" sets all signals to their asserted value; that is, low asserted signals become 0 and high asserted signals become 1. "-*" sets all signals to their non-asserted values.

MEM_INIT { 0 |1 |* |-* |U }

Initializes the contents of memories to the specified values. U is only a legal option if the "MEM_STATE 4;" directive has been given.

MEMLOAD *file name*[,*file bit range*,[*file word range*], *primitive bit range*,[*primitive word range*]]

Loads the memory specified by the current Memory path from *file name*. Note that the square brackets around the file and primitive word ranges are required. The user is prompted for a file name if none is given. All other parameters are optional. The optional bit and word ranges specify a mapping from the memory contents file to the memory primitive. Note that all of the ranges are taken to be in decimal regardless of the current radix. See the Loading Memories section.

MEMPATH *pathname*

Sets the "Memory path" part of the status line to *pathname.* Note that pathname must be the pathname of a memory primitive and must be enclosed in parentheses. The pathname need not be complete, but must uniquely define a primitive. Memory pathnames are necessary in order to display or change memory locations or load memories from files. If no memory can be found that matches the given pathname, the memory that best matches the pathname is used. The NEXTMEMORY command can be used to advance the mempath to another memory.

MOVE *from_point to_point*

Allows the user to change the position on the screen of a previously OPENed signal. *from_point* and *to_point* are specified using the puck in the Simulator window. The signal currently being displayed at *from_point* will be removed and redisplayed at *to_point*, replacing any signal which may already be at that location. This command is only available where puck usage is enabled (GED or graphics Simulator).

NEXTMEMORY

Advances the mempath to another memory.

OPEN *signal* [, *row* [, *col*]]
or
OPEN *signal pt* [(*dest pt signal pt*)...] [*dest pt*]

Opens a signal (i.e., adds a signal to the display). Note that the second syntax is only available where puck usage is enabled (GED or graphics Simulator). *signal pt* is the puck point that identifies the signal in the drawing or in the lower window; *dest pt* is the point that defines where the signal is to be displayed. If *dest pt* is omitted, the Simulator opens the signal

in a default location, usually as near as possible to the top of the display. The sequence of (*dest pt signal pt*) can be repeated; the command must be terminated by a semicolon or carriage return. In WAVEFORMS mode, opening a signal also causes its history to be recorded.

If a signal has not already been opened and empty rows remain on the current screen, omitting *row* causes the signal to appear in the first free row. If the screen is filled, the next available row (not on the screen) is used, and the display is shifted to display this signal. If the same signal was previously opened and no position is indicated, the existing signal is marked as open (shifting, if necessary, to display it).

The user can replace an existing signal by opening a new signal and specifying *row*. Once a signal is Opened in WAVEFORMS mode, the history for the signal is maintained for the specified history period, even if the signal is not on the screen (i.e., a user can Open more signals than can be displayed at one time. Simulate to calculate their behavior, and then view their behavior).

OPENMEMORY ADDRESS [, row [, column]]

Adds the contents of the addressed memory location to the main display. The memory is identified by the pathname last given to the MEMPATH command. The pathname appears in parentheses, followed by the address in parentheses. The memory word appears in the current radix, and becomes the current signal for purposes of depositing a new value. The address uses the current radix and must not contain any bits with value U (unknown) or Z (high impedance). If a location (row or row, column) is specified, the memory display is placed at that location if it is available. The column argument is only used in Bus mode.

PAUSE

Stops taking commands from the current command file and returns control to the terminal. The RESUME command returns control to the command file.

PEEK *signal*

Allows the user to observe the value of a specified signal without requiring that it first be OPENed in the display area. The signal value is simply output in the echo area in the current radix. *signal* may be specified using the puck. If CURSOR time is something other than the current time, the command will output the value of the specified signal at both the CURSOR time and the current time. If the specified signal has no history, the message in the echo area will so indicate and the command will output its current value.

PERIOD *value*

Sets the clock period to the specified decimal integer value (*value*). The clock period is displayed in the status lines. If the specified period is too small, a warning is given.

PLOT [*start_time end_time*] ['*filename*']

Builds a waveform diagrams file that can be used as input to the PLOTTIME program to produce waveform diagrams under GED. The default parameters are the waveform starting time, waveform ending time, and the file name 'plotsig.dat', respectively. Note that specifying *filename* closes any previously-specified file and opens a new file for output. Also note that after invocation, the file is not closed and subsequent calls without the *filename* parameter append additional data onto the previously opened file. For additional information on the Plottime

program, see The Plottime Reference Manual.

RADIX { 2 | 8 | 10 | 16 | B | O | D | H | S }

Sets the current radix. The radix value may be either 2 or B for binary, 8 or O for octal, 10 or D for decimal, 16 or H for hexadecimal, or S for strength. The default radix is hexadecimal.

RECORD_ALL

The RECORD_ALL command causes the signal histories of all signals AND all memories in a circuit to be recorded. This command is identical to the RECORD_SIGNALS command (see below) except that the history of all locations of all memories also is recorded. Note that considerable storage requirements could be involved in creating and maintaining a history of all signals and memories. Thus, this command should not be invoked on circuits with a large number of elements and/or large memories.

RECORD_SIGNALS

Causes the signal histories of all signals in the circuit to be recorded. By invoking this command the history of all signals is available thereafter.

Note that the RECORD_ALL command does not affect the duration of history that is maintained for all signals. Also note that since certain storage requirements are involved in creating and maintaining history, this command should not be invoked on large circuits.

REDISP

Erases the screen and redraws the status lines and main display. The echo area disappears.

REMOVE [*signal*]

Removes the indicated signal from the signal display area. *signal* is an optional parameter which may be specified using the puck. If *signal* is not specified, the currently OPEN signal is REMOVEd. If *signal* is entered from the keyboard, a single occurrence of the signal in the current radix is REMOVEd. When selected from the menu, the user is prompted for a signal name.

RESUME

Restores the SAVEd status of the Simulator from the file *filename.* Returns command control to the command file at the point of the most recent PAUSE or COMPARE command.

ROW *top row number*

Specifies signals to be displayed by defining the signal to be positioned at the top of the display in the WAVEFORMS mode. See the Waveforms section.

SAMPLE ENABLE *signal* GETS *expression 1* WHEN *expression 2*

Samples an ENABLE signal to *expression 1* when *expression 2* becomes true. See the Breakpoint section.

SAMPLE PATCH *signal* GETS *expression 1* WHEN *expression 2*

Samples a PATCH signal to *expression 1* when *expression 2* becomes true. See the Logic Patching section.

SCOPE *pathname*

Defines a default *pathname* to be used for signal identification. If the user sets the scope to the desired drawing or part, signals can be identified without having to type the pathname. (Note that even without defining scope, the Simulator accepts an abbreviated or missing pathname if it uniquely specifies a signal.)

SCRIPT *file name*

Changes the input stream so that the Simulator reads from the specified file. The file name does not need to be in quotes, but must follow the file name conventions of the host machine; for example, in UNIX, the case of letters is significant, while in VMS, it is not. The Simulator echos the commands in the script file at the terminal, but does not prompt.

A convenient way to create a script file is to edit the Simulator output file simcmd.dat (command file) and rename it. See also the PAUSE and RESUME commands for further information on script files.

SCROLL

This command controls the automatic scrolling feature of the Simulator. The command takes the values ON or OFF. The default is ON. See the Waveforms section.

SET BREAKPOINT *expression*

Installs *expression* as a breakpoint. See the Breakpoints section.

SET BREAKPOINT # *number*

Activates a numbered breakpoint. The number must be preceded with the # character. See the Breakpoints section.

SET ENABLE *signal* WHEN *expression*

Sets an ENABLE signal when *expression* is true. See the Breakpoints section.

SET { Local_plot | Spooled_plot }

Specifies what to do with HARDCOPY output; LOCAL_PLOT queues the output immediately, while SPOOLED_PLOT sends the output to a file for output using the HPR utility. LOCAL_PLOT is the default. The spool file has the name 'hard' when there is only a single window, or 'hardXY' (XY is the tty number of the current window) on systems having multiple windows. This command is only available with the full-screen Simulator. It has no effect when running the split-screen simulator. See the HARDCOPY command.

SET PATCH *signal* WHEN *expression*

Sets a PATCH signal when *expression* is true. See the Logic Patching section.

SET

The SET command specifies the plotter type for HARDCOPY output. The allowed arguments are:

W11versatec
W22versatec
W36versatec
W42versatec
Calcomp1043
Calcomp5744
B9424
HP7475
HP7580

The same plotter types are supported as in GED with the same name specifications. This command is only available with the graphics Simulator - it has no effect when running under GED. The 11" Versatec is the default. See the HARDCOPY command.

SHOW *pathname*

Accepts a pathname in the same format as the MEMPATH command and displays the current values of all the signals connected to the primitive at that pathname. This command is most useful for testing simulation models during library development.

SIMULATE { val |C |S } [*display percentage*]

Simulates and advances simulated time by the specified number of nanoseconds. If the command "SIMULATE C" or "SIMULATE S" is given, time is advanced by one clock period or one step, respectively. When simulating past the final time displayed on the screen in WAVEFORMS mode, the display automatically shifts to display a new interval. The optional *display percentage* parameter indicates the percentage of the screen width which is to be occupied by waveforms when this shift occurs. The default is 50 percent of the screen but any value from 0 to 100 may be specified.

SNAPSHOT

Prints an image of the status lines and signal display window in the List file if a List file is being created.

SPACING *value*

Sets the spacing between adjacent waveforms in WAVEFORMS mode to *value*. The default value is 1, indicating single spacing. See the Waveforms section.

STEP *value*

Sets the simulated time step size to the specified decimal integer *value*.

SYSTEM

Returns user temporarily to operating system (UNIX or VMS) so that they may give a command. Useful for editing command files and script files.

The Simulator prompts for an operating system command, executes the command and redisplays the simulation display. The operating system command can also be given as an argument to the SYSTEM command like this:

```
system ls
```

The SYSTEM command is not supported in the split-screen Simulator. Instead, go into GED and use the UNIX or SYSTEM command.

TERMINAL

Sets the terminal type. Accepted arguments are:

ANNARBOR	An Ann Arbor Ambassador terminal set to 48 lines.
CLUSTER	A SCALD terminal with character graphics, running locally or in transparent mode connected to the host computer.
GCLUSTER	A SCALD terminal with graphics capabilities enabled.
GRAPHICS	An IBM PC terminal or VAXstation II terminal.
TTY	Any video terminal or a teletype (this is the default).
VT100	A DEC VT100 with 24 lines.
3270	An IBM 3270.

TRACE *signal*, [*radix*]
or
TRACE *pt* [*pt* ...]

Traces the output or outputs corresponding to the given signal or signal subrange. The second syntax indicates that *signal* may be specified using the puck to point at it. *radix* is an optional parameter that may be specified using numerals (2, 8, 10, or 16) or characters (b, o, d, or h). If no radix is specified the default trace radix is used. See the Tracing section and the TRACE_RADIX and LIST TRACES commands.

Example: * trace data
* trace out<66..33>,h
* trace midout<4>

TRACE_ALL

Traces all outputs of all signals and all contents of all memories.

TRACE_CLOSE

Closes all trace output files. Usually means that all tracing for the current simulation is complete.

TRACE_INTERVAL *number*

For Tabular I/O format, causes a trace record to be output every *number* nanoseconds during the simulation. *number* must not be less than 0. If *number* is 0 (the default), a trace record is written every time there is at least one transition. This command is ignored when the standard trace format is being used.

TRACE_MEM

Traces the contents of the memory currently specified by the MEM_PATH command. This command only works for standard tracing.

TRACE_OPEN

Opens the trace output file(s). If the simulation is using the standard trace format, the signal mapping file is output when this command is given.

TRACE_RADIX [2 |8 |10 |16 |b |o |d |h]

Changes the default radix used for tracing (initially set to 2). If no parameter is specified, the current default trace radix is output.

TRACE_READ *filename*

Reads in a Tabular I/O trace file from a previous run (or manually generated) to stimulate the circuit. The signals to be traced are first read in, followed by the list of times and signal values. As each time is reached in the simulation, the values for that time are deposited into the proper signals. To see the values being deposited as the simulation advances, use the UPDATE_INTERVAL command.

TRACE_RESET

Resets (disables) stimulation from a tabular input file. This command can be given at any time to turn off circuit stimulation. A different tabular input file may also then be specified. See the Tracing section.

TRACE_START

Begins tracing the outputs specified by either the TRACE_ALL command or appropriate TRACE command. TRACE_START can be given any number of times during a simulation run. See the TRACE_STOP command.

TRACE_STOP

Discontinues tracing until another TRACE_START command is entered.

UNDO DEPOSIT { *signal* | *pt* }

Cancel all future scheduled DEPOSITs for a specified signal. *signal* may be specified using the puck. If there are no outstanding DEPOSITs for a given signal for time > current time, a message will be output. Note that this command is intended to undo previously issued DEPOSIT commands made to a specific signal; thus, attempts to UNDO DEPOSITs to a superset, subset, synonym, etc. of the signal will

fail (unless a DEPOSIT was scheduled for that signal as well). For example, given the following sequence of commands at time=0:

```
dep a<7..0>, 45@100 1b@110
dep a<3..0>, 7@100
undo dep A<3..0>
```

The 'UNDO DEPOSIT' has no effect on the first DEPOSIT command, and the value 45 will still be scheduled for output at time=100.

UPDATE_INTERVAL *constant*

Sets the simulator to update the screen at specified intervals while simulating for a longer time. If zero is specified, any previously set interval is cleared and updating is disabled.

WAVEFORMS { *start time* { *end time* |; } |; }
or
WAVEFORMS *pt1* { *pt2* |; }

Enters WAVEFORMS mode with the specified parameters. See the Waveforms section.

WIRE_DELAYS *filename*

Specifies the name of the wire delays file. See the section on Delays.

WRITE_COVERAGE *filename* [, { 0 |1 |2 |3 }]

Outputs the list of signals that have made a transition and the number of transitions that they have made. If the optional parameter (0 - 3) is specified, the signals are processed based on the number of times that they have made a transition. The signals are sorted by the number of transitions, and the file only contains those signal names in specific groups; for example, specifying "0" indicates that only signals

making 0 transitions (i.e., those that have not changed) should be output, and "1" indicates that only those signals making 0 or 1 transitions are output. See also the COVERAGE and INIT_COVERAGE commands.

SECTION 8
DELAYS

When making simulation models, delay values can be added to each primitive using the body properties DELAY, RISE, and FALL. Pin-to-pin delays can be specified using the pin properties PDELAY, PRISE, and PFALL. For designs where delays are related to changes in output loading, temperature, and voltage, the Delay Estimator and Expression Evaluator can be used. Each of these features is described below.

8.1 DELAY PROPERTIES

In most simulation models delays are modeled using three body properties: DELAY, RISE, and FALL.

DELAY PROPERTY

The DELAY property accepts two values, a rise delay followed by a fall delay and separated by a comma. If only one value is specified, this value is used as both the rise and fall delay.

All of the values for delay properties can be specified as a single number, or as three numbers enclosed in square brackets. When three numbers are used, the first value is the minimum delay, the second is the typical delay, and the third is the maximum delay. For a particular simulation run, you select to use either all of the minimum values, or all of the typical values, or all of the maximum values. The DELAY_MODE directive is used to select one of the three values for the current simulation run. Thus, delay can be specified in one of the following formats:

DELAY *delay time*
DELAY *rise delay, fall delay*
DELAY [*min,typ,max*]
DELAY [*min,typ,max*],[*min,typ,max*]

RISE AND FALL PROPERTIES

In addition to using the DELAY property, rise and fall delays can be specified using the RISE and FALL properties. Usage of these properties is as follows:

RISE *rise delay*
FALL *fall delay*

When the DELAY property is attached to a body where the RISE and/or FALL properties are also attached, the RISE_FALL directive is used to select which delay values are used. See the Directives section for more information.

PIN-TO-PIN DELAY

For complex primitives with multiple input and output pins, accurate modeling of the delays within the primitive is tedious if not impossible. The pin-to-pin delay feature allows the user to associate separate delay values for individual paths from input to output pins.

To use pin-to-pin delays, you use the pin properties PRISE, PFALL, and PDELAY. These properties designate respectively pin rise delay, pin fall delay, and pin delay. For greatest flexibility, these properties are permitted on both input pins and output pins.

When the delays through a primitive are all one set of values except for one delay path, the one unusual delay path can be specified using the pin properties and the other delays can be specified using the body properties. The attributes for the properties are: permit(pin), inhibit( ).

The property values must be in the format described below:

PRISE = *PV list*
PFALL = *PV list*
PDELAY = *PV list*

<PV list> ::= <PV> |<PV> , <PV list>
<PV> ::= <P> : <V>
<P> ::= <pin name> |(<pin list>)
<pin list> ::= <pin name> |<pin name> , <pin list>
<V> ::= <delay value> |[<triple values>]
<triple values> ::= <min> , <typ> , <max>
<min> ::= <delay value>
<typ> ::= <delay value>
<max> ::= <delay value>
<delay value> ::= <empty> |<value>

To enable the pin-to-pin delays the PIN_DELAY directive must be set to ON. OFF is the default value for the PIN_DELAY directive. The pin delay properties override the body delay properties when the PIN_DELAY is set ON; otherwise, the body delay properties override the pin delay properties. When the PIN_DELAY directive is set to ON, the body properties DELAY, RISE, and FALL are used for the pairs of input and output pins where pin delays are not specified.

Any conflicts between input pin properties and output pin properties (that is, two different delay values specified for one pin-to-pin path) is reported as an error. When delay values are specified between two input pins or two output pins, an error is also reported.

RISE_FALL DIRECTIVE

The RISE_FALL directive is used to control the use of separate RISE/FALL delays for both parts and pins. By default, this directive is ON. See the Directives section for more information on this directive.

8.2 EXAMPLES USING DELAY PROPERTIES

Here are two examples showing the RISE_FALL directive and the PIN_DELAY directives working together.

PIN DELAYS WITH RISE/FALL DELAYS

The ADDER primitive has inputs A, B, PI, GI, and CI, and outputs F and CO. For this example both the PIN_DELAY and the RISE_FALL directives have the values ON.

The ADDER body has the DELAY property attached with these values:

DELAY = [3,4,5]

The input pin A has the following pin properties:

PDELAY = (F):[2.5, 3.7, 4.4]

PRISE = (F):[2.3, 3.4, 4.5], CO:[1.1, 2.4, 3.5]

PFALL = (F):[1.1, 2.2, 3.3], CO:[2.1, 3.2, 4.3]

For all delay paths from the input pins B, PI, GI, and CI the values of the DELAY property are used.

For the delay paths from A to F and from A to CO the following delay values are used:

		MIN	TYP	MAX
A to F	(rise)	2.3	3.4	4.5
	(fall)	1.1	2.2	3.3
A to CO	(rise)	1.1	2.4	3.5
	(fall)	2.1	3.2	4.3

PIN DELAYS WITHOUT RISE/FALL DELAYS

The REG RS primitive has inputs I, CK, R, and S, and output Q. Assume that the PIN_DELAY directive is ON and the RISE_FALL directive is OFF.

The REG RS body has the DELAY property attached with these values:

DELAY = [3,4,5]

The output pin Q has the following pin properties:

PDELAY = (R):[2.5, 3.7, 4.4]

PRISE = (R,S):[2.1, 3.5, 4.0], CK:[1.1, 2.4, 3.5]

PFALL = (R,S):2.2, (CK):[0.9, 2.5, 4.3]

For the delay paths from I, CK, R, and S to Q the following delay values are used:

	MIN	TYP	MAX
I to Q	3	4	5
CK to Q	1.1	2.5	4.3
R to Q	2.5	3.7	4.4
S to Q	2.2	3.5	4.0

8.3 DELAY ESTIMATOR

In many technologies, the time required for the output of a component to reach its loads is affected by both the interconnect delay and the size of the load. If Tr is the calculated rise time and Tf is the calculated fall time, then:

Tr = Tdr + Kr(load on the net) + Tir
Tf = Tdf + Kf(load on the net) + Tif where

Tdr	is the component's rise delay
Kr	is a device-specific constant related to changes in the output's rise time as a function of component loading
Tir	is the rising edge delay due to wires
Tdf	is the component's fall delay
Kf	is a device-specific constant related to changes in the outputs fall time as a function of component loading
Tif	is the falling edge delay due to wires
Note:	Tdr, Tdf, Kr, Kf are device-specific; (load on the net) is net specific; and Tir, Tif input specific.

We can lump together the net loading term and interconnect term of the delay. Then the delay due to all interconnection effects can be modeled as an input specific wire delay. If interconnection delays are computed (or estimated) this way by the physical design system, and then fed back to the Simulator as wire delays (in the delay file specified with the WIRE_DELAYS directive), we obtain an accurate representation of the system. Early in the design cycle however, it may be impractical to provide such detailed delay information -- estimators are required.

The Simulator has a Delay Estimator that takes into account static load, and provides a wire delay estimate based on the number of stops (inputs and outputs) on the

net. This delay estimate is added to the basic component delay (specified with the RISE, FALL, and DELAY properties in the simulation model).

To use the Delay Estimator you need to use the directive DELAY_ESTIMATOR ON and set the other appropriate directives to the required values.

The Delay Estimator uses the following equation:

Tr(estimated) = Tdr + Kr(loads on the net + wire delay)
Tf(estimated) = Tdf + Kf(loads on the net + wire delay)

where constants Tdr, Kr, Tdf, Kf are as above.

The load term is a weighted sum of inputs and outputs on the net which approximates the true capacitive and DC load on the net. The wire delay is estimated by counting the number of stops on the net and converting stops into load equivalents.

INTERACTION OF WIRE DELAYS WITH DELAY ESTIMATOR

If a delay file is used to feed back delay information from a physical design system and the Delay Estimator is also used, the values from the delay file override values calculated by the Delay Estimator.

COMPUTING NET DEPENDENT DELAYS

The Simulator estimates the net delay on each net in six steps:

1. The load is estimated by taking a weighted sum of the inputs and outputs on the net.

2. The number of stops on the net is counted.

3. The number of stops is converted to an interconnection delay estimate (in units of load equivalents) by table look-up.

4. An effective net load is computed by adding the interconnect and load estimates.

5. The effective net loading is multiplied by the drive constants (Kr and Kf) of the drivers of the net to obtain rise and fall delays due to net loading.

6. These delays are added to the drivers zero-load parameters (Tdr and Tdf).

Counting Loads

Counting the inputs and outputs on a net is complicated by the presence of TIMES properties and dots.

When a net is connected to the output pin or input pin of a library part that has the TIMES property attached to it, the value of the TIMES property affects the number of loads on the net.

When a net is connected to one or more input pins and a single output pin, the following rule applies:

> Each input pin is counted *n* times when TIMES = *n* and the sum of the values of the TIMES properties for each pin is used. The output pin to which the net is connected is counted once and when the output pin is connected to a library part having a TIMES property, the total number of inputs on the net is divided by *n*.
>
> For example, a net is connected to an output pin of a library part with the property TIMES=3, and to three input pins. The input pins are on three different library parts that have, respectively, the properties TIMES=5, TIMES=2, and TIMES = 1. The load is calculated as 5 + 2 + 1 = 8 inputs on the net. This total number of inputs is divided by the value of the TIMES property of the output pin. This gives 8 divided by 3.

When a net is connected to several output pins that are dotted together (connected together) and the output pins belong to library parts that have the TIMES property

attached to them, the previous rules apply with two changes:

> The number of output pins on the net is the sum of each output pin ignoring the TIMES properties. The number of inputs on the net is counted as before, and then divided by the smallest value of the TIMES property of any library part with an output on the net.
>
> If phantom gates are used, they are collapsed to an explicit dot for the counting procedure.

The user may , in addition, place an optional pin property, LOAD_FACTOR, on any pin. LOAD_FACTOR takes a fixed point number as a value. If LOAD_FACTOR is specified, a pin is counted LOAD_FACTOR times (or LOAD_FACTOR * *n* times when it is an input pin and a TIMES property with a value of *n* is present) rather than once in the above counting procedure.

Estimating Wire Delays

To estimate wire delay, the number of stops on each net is counted. If phantom gates are used, they are collapsed into an explicit dot for the stop counting process. The number of stops is converted to equivalent loads by table lookup using a table specified with the Simulator directive WIRE_ESTIMATE. WIRE_ESTIMATE takes an argument list of fixed point numbers and an optional FAMILY specification. A net with *j* stops receives a wire delay estimate given by the *j*th number in the list. The family specification allows for a number of different WIRE_ESTIMATE tables to be used in the same simulation run. If a FAMILY body property is given on a primitive, then the WIRE_ESTIMATE table with the same FAMILY specification will be used. If no FAMILY body property is given on a primitive, then the WIRE_ESTIMATE table without a FAMILY specification will be used. An example set of WIRE_ESTIMATE directives are given below:

```
WIRE_ESTIMATE 1.0, 2.0, 3.0, 4.0;
WIRE_ESTIMATE ECL: 0.5, 1.0, 2.0, 3.0;
WIRE_ESTIMATE TTL: 1.0, 2.0, 3.1, 4.0;
WIRE_ESTIMATE ON_GATE_ARRAY: 0.3, 0.6, 1.0, 1.3;
WIRE_ESTIMATE BET_GATE_ARRAY: 1.0, 2.0, 3.1, 4.5;
```

Computing Load Dependent Net Delays

In simulation models, each primitive that drives an output pin of the part being modelled can have an optional pin property, DRIVE. This property takes a pair of values, the first value is the driver's Kr factor, the second its Kf factor. For details of the syntax for the DRIVE property, see below under Expression Evaluator. If no DRIVE property is specified, Kr and Kf are set to the default value specified by the DEFAULT_DRIVE directive. When neither DRIVE properties nor the DEFAULT_DRIVE directive are used, Kr and Kf are set to 0-0. If only one value is given, Kr and Kf are both set to that value.

After the effective net loading has been computed for a component's output net, the component's output delays (DELAY, or RISE/FALL) are adjusted **on a bit-by-bit basis** by the time obtained by multiplying its drive constant(s) by each output bit's effective net loading.

USING THE DELAY ESTIMATOR

To use the delay estimator, follow these steps:

1. Use the DELAY_ESTIMATOR directive to turn on delay estimation. The default value for this directive is OFF.

2. Specify drive constants (Kr, and Kf). This can be done in two ways:

    - By attaching the DRIVE pin property to each Simulator primitive whose output is to display load-dependent behavior.

- By specifying a default value for drive constants. To do so, use the DEFAULT_DRIVE directive. See under Directives for details.

3. Specify pin loading. This is done by attaching the LOAD_FACTOR property to a pin of the Simulator primitive that connects to the interface signal that represents the pin of the part whose load is to be specified. The LOAD_FACTOR property takes a fixed point number as its value:

   LOAD_FACTOR = *fixed point number*

   If no LOAD_FACTOR property is specified, a default LOAD_FACTOR value of 0 is used.

4. Specify a conversion table from stops to load equivalents. This is done with the WIRE_ESTIMATE directive. See under Directives for details.

Assuring Correct Load Counting

This scheme for counting stops and loads on a net is independent of the actual wiring of a net. In two significant cases this results in delay estimates that are too large.

Drivers with the TIMES property, especially those feeding wire gates or phantom gates, are often wired with a careful partitioning and placement of the loads. The estimation scheme does not take this into account. It assumes a load that results from an even partitioning of the loads into a number of pieces equal to the smallest value of the TIMES properties found on the drivers.

Physical parts often have common input pins which are modelled as separate pins. For example in a design that uses two LS374s, each with the property SIZE=4, both parts could be driven by the same clock signal and hence be allocated to the same package. However, since two logical parts appear on the GED drawing, two LS374 clock pins will appear on the net instead of one.

8.4 EXPRESSION EVALUATOR

An expression evaluator feature allows the user to specify equations to adjust primitive delay times based on output loading, temperature, and voltage variation. Several directives and properties are used to implement this feature. If the Expression Evaluator and the Delay Estimator are used simultaneously, the Simulator uses the values produced by the Expression Evaluator.

Drive and load factor can be specified as pin properties in drawings using the properties DRIVE and LOAD_FACTOR; both of these properties can take real parameter values. The body properties DELAY_EQ and DELAY_PARAM allow the user to specify equation names and parameter names to be used in calculating delay. The TIMES property may be specified on bodies in drawings to be simulated. The syntax of these properties is as follows:

DRIVE = <drive> or
DRIVE = <rise_drive>, <fall_drive>
LOAD_FACTOR = <real>
TIMES = <integer>
DELAY_EQ = <eq_id> or
DELAY_EQ = <rise_eq_id>, <fall_eq_id>
DELAY_PARAM = <param_id> or
DELAY_PARAM = <rise_param_id>, <fall_param_id>

The expression evaluator is controlled in the Simulator with several directives. The EXP_EVALUATOR directive controls whether the feature is used in the Simulator. Default rise and fall drive values can be specified using the DEFAULT_DRIVE directive. Finally, users can define delay equations containing variables via the USER_PARAMETER and USER_EXPRESSION directives. See the Directives section for more information on these directives.

Here is an example explaining the operation of these properties and directives. Say that three equations define the delay function for parts in a certain gate array library:

1. LOAD <= COF
   tpd = t0 + DRIVE * LOAD

2. COF < LOAD <= 2*COF
   tpd = t0 + 1.5*DRIVE*LOAD - 0.5*DRIVE*COF

3. 2*COF < LOAD <= 4*COF
   tpd = t0 + 3*DRIVE*LOAD - 3.5*DRIVE*COF

where the variables are defined as follows:

LOAD	output node loading
COF	cell output factor which differs from cell to cell
DRIVE	cell output drive capability
tpd	overall cell delay time including loading effect
t0	cell delay time excluding loading effect

In the design drawing a body named INVERTER is used and two body properties are attached to it:

DELAY_EQ=DelayEq
DELAY_PARAM=INV1

Delay Eq is a property value and gives the name of the delay equation, or as we call it, the equation identifier. DelayEq is chosen by the user. The output pin of the INVERTER has the following pin property attached to it:

DRIVE=0.35

The equation identifier, DelayEq, and parameter identifier, INV1, must be defined in the Simulator Directives file so those references can be resolved. The equation identifier is defined in the USER_EXPRESSION directive. The parameter identifier is defined in the USER_PARAMETER directive. The USER_EXPRESSION directive can be used to define the following delay equation:

```
USER_EXPRESSION DelayEq(Cof)=
    [lu<=Cof]'drive*lu',
    [Cof<lu<=2*Cof]'1.5*drive*lu-0.5*drive*Cof',
    [2*Cof<lu<=4*Cof]'3*drive*lu-3.5*drive*Cof';
```

where:

DelayEq	is the equation identifier
Cof	is the value given in the USER_PARAMETER directive applicable to this primitive (different values can be used for different cells).
lu	is the loading value calculated by the Simulator from the value of the LOAD_FACTOR property.
drive	is the value of the DRIVE property attached to the cell output pin.

The USER_PARAMETER directive is used to define the value of Cof as follows:

```
USER_PARAMETER INV1=18;
```

The value 18 is then substituted for Cof in the equations for each library part that has the property DELAY_PARAM = INV1 attached. When two different USER_PARAMETER directives are used, for example,

```
USER_PARAMETER INV1=18;
USER_PARAMETER INV2=10;
```

the value 18 would be substituted for Cof in the equation for the library parts having the property DELAY_PARAM = INV1 and the value 10 would be substituted for Cof in the equation for the library parts having the property DELAY_PARAM = INV2.

To return to the example, if the total number of load units (lu) at the output of the INVERTER is 20, the Simulator can evaluate USER_EXPRESSION. The second equation is selected since 18 < 20 < 2*18, and the computation proceeds as follows:

$$\begin{aligned} \text{DelayEq}(18) &= (1.5 * 0.35 * 20) - (0.5 * 0.35 * 18) \\ &= 7.35 \end{aligned}$$

The computed delay, 7.35, is added to the delay for this INVERTER.

8.5 WIRE DELAY FEEDBACK

After a design has been sent to a physical design system and layout and routing performed, precise wire delay information is available. This delay information can then be fed back into the Simulator and the design can be resimulated to verify that it still performs as required.

WIRE DELAY FILE

The delay information is read by the Simulator from the wire delay file. The wire delay file can have any name. You specify the name of the wire delay file with the WIRE_DELAYS directive or with the WIRE_DELAYS command. Do not use the wire delay file when using Realfast.

The format for the Simulator wire delay file is similar but not identical to the format of the Timing Verifier delay.dat file. The wire delay file must be in the format described below. Each element in the file consists of a signal name in quotes, a bit subscript (if any), and a delay element or a list of path names of components that the signal drives with a delay for each path. When multiple path names are specified, the Simulator chooses the minimum delay from all of the specified delays for each path. These delays are added in with any other specified delay values to determine when Simulator events should be scheduled for those outputs.

Ranges cannot be specified in the Simulator delay file. Minimum, typical, and maximum delay values cannot be specified in the Simulator delay file.

```
<delay file>        ::= END. |
                        <delay list> ; END.

<delay list>        ::= <signal delay list>; |
                        <signal delay list> ; <delay list>

<signal delay list> ::= <signal name> : <stop delay list>

<stop delay list>   ::= <stop delay>; |
                        <stop delay>, <stop delay list>;

<stop delay>        ::= = <quoted rise/fall>; |
                        <quoted path name> = <quoted rise/fall>;

<signal name>       ::= <quoted signal name> |
                        <quoted signal name> < <bit range> >

<quoted signal name> ::= '<name>'

<bit range>         ::= <bit number> |
                        <bit number> .. <bit number>

<bit number>        ::= <integer>

<quoted path name>  ::= '<path name>'

<quoted rise/fall>
                    ::= '<delay>' |
                        '<rise delay>, <fall delay>' |
                        '<rise delay> - <fall delay>'

<rise delay>  ::= <delay>

<fall delay>  ::= <delay>

<delay>             ::= <fixed point number>
```

Here is an example of a wire delay file:

```
'DATA' <5..0>: = '2.3, 3.4';
'ENABLE' : = '5.1';
'(SYS ALU MUX)' = '2.3,3.4',
'(SYS REG)' = '1.7-1.2';
END.
```

Notice that the minus sign in the rise/fall specification does NOT indicate a range of values. The first value specifies the rise time and the second value specifies the fall time.

USING A WIRE DELAY FILE

Once the wire delay file is created, you instruct the Simulator to use the file either with a directive in the Simulator Directives file, or with an interactive command.

The directive you use is:

WIRE_DELAYS '*name* ';

We suggest you use a name other than delay.dat to avoid confusion with the Verifier delay.dat file.

The command you use is:

WIRE_DELAYS *name*

SECTION 9
SIMULATION MODELS

As part of each Valid-supplied library, there is a simulation model for each component in the library. The simulation model simulates the functionality of the component and is used by the Logic Simulator. Simulation models are built by library developers from a specific set of parts called Simulator primitives that are contained in the library named SIM. A wide variety of Simulator primitives are available from simple logic gates to a complete ALU. The behavior of each primitive is understood by the Logic Simulator.

The model is entered into the SCALDsystem as a GED drawing having the extension .SIM. For example, the simulation model of an LS74 is a drawing with the name LS74.SIM. It can be viewed on the screen using the EDIT command. Permissions are usually set on component models so that only the librarian or root has permission to change the models.

9.1 USING SIMULATION PRIMITIVES

Each input and output pin on a primitive may be individually bubbled using the Graphics Editor command "BUBBLE." A bubbled pin has an intrinsic inversion; that is, an AND gate with a bubbled output behaves as a NAND gate. The function table for a bubbled AND gate is:

Input	Output
0	1
1	0
Z	Z
U	U

A Simulator primitive can have a SIZE property to specify its bit width. For example, to compute the sum of two 16-bit signals, a single adder primitive with a SIZE of 16 can be used, instead of 16 adder primitives. Two special primitives, the "8 BIT PRIO ENCODER" and the "1 OF 8 DECODER" have a fixed SIZE of eight bits. A primitive may be given a size of "SIZE" which means that the size of the primitive is taken from the size property of the part being modeled. Many primitives have inputs and outputs that are not affected by the size property. All enable inputs, clock inputs, and chip select inputs have a fixed width of one bit. The select input of an 8-bit multiplexer is always three bits wide.

Simulator primitives may be given a DELAY property. Primitives without an explicit DELAY are assumed to have a delay of 0. Delays are given in nanoseconds. By convention, primitives are given delays to model the worst-case behavior of the part being modeled, but this is not required. The SCALD Timing Verifier uses a different set of timing models. For the Simulator to function correctly, it is sufficient that the timing behavior of the Simulator model represents one possible timing behavior of the part. The user should exercise care when specifying delay values for parts; in particular, zero-delay parts may result in unexpected behavior in a circuit.

Logic Simulator models must include interface signals with names that correspond to the names of the signals in the body drawing for the part being modeled. A DRAWING body and a DEFINE body should be included in each Logic Simulator model. The DRAWING body should be given a TITLE property and an ABBREV property. The TITLE should be the name of the body and the ABBREV should be an abbreviation that is easily recognizable.

9.2 SIMULATOR PRIMITIVES

The various Logic Simulator primitives are described in this section. Function tables are included to document the behavior of primitive outputs. The * (asterisk) character is used to designate low assertion on inputs pins and to complement output pins.

9.3 LOGIC GATE PRIMITIVES

There are three types of logic gate primitives: AND, OR, and XOR. Since any pin of any primitive may be independently bubbled, to create a NAND gate, simply bubble the output of an AND gate.

AND PRIMITIVE

The AND primitives come in seven varieties, 2-input through 8-input: 2 AND, 3 AND, 4 AND, 5 AND, 6 AND, 7 AND, and 8 AND. The truth table for an AND primitive is:

One or More Inputs	All Other Inputs	The Output
0	X	0
1	1	1
Z,U	1	U

where X is any value.

OR PRIMITIVE

The OR primitives also come in seven varieties: 2 OR, 3 OR, 4 OR, 5 OR, 6 OR, 7 OR, and 8 OR. The truth table for an OR primitive is:

One or More Inputs	All Other Inputs	The Output
0	0	0
1	X	1
Z,U	0	U

where X is any value.

XOR PRIMITIVE

The XOR has only a 2-input version. The truth table for an XOR primitive is:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0
X	U,Z	U
Z,U	X	U

where X is any value.

9.4 BUFFER PRIMITIVES

There are three buffer primitives: the simple buffer BUF, the tri-state buffer TS BUF, and the identity buffer, IDENTITY.

BUF PRIMITIVE

The truth table for the BUF primitive is:

Input	Output
0	0
1	1
Z,U	U

To create an inverting buffer, simply bubble the input or output pin of a buffer. Non-inverting buffers are commonly used for delays.

TS BUF PRIMITIVE

The tri-state buffer primitive TS BUF has an enable input which, when disabled, causes the output to take the value high impedance (Z). The enable input has a width of one bit. Here is the truth table for the TS BUF:

Input	Enable*	Output
0	0	0
1	0	1
Z,U	0	U
X	1	Z
X	Z,U	U

IDENTITY

The IDENTITY primitive is similar to BUF except that it propagates the exact signal on the input pin to the output pin, while the BUF primitive converts the Z state to U and soft values to hard values. Here is the truth table for the IDENTITY primitive:

Input	Output
0	0
1	1
Z	Z
U	U

9.5 JK PRIMITIVE

The JK primitive models the J-K Flip Flop. The primitive has input pins for J and K data inputs, asynchronous set and reset functions, and an edge-sensitive clock. If the clock input is not bubbled, then the primitive's outputs triggers on a positive edge; if it is bubbled, it triggers on a negative edge. Outputs consist of Q and Q-BAR data outputs. Asserting both the set and reset pins causes both of the outputs to go high. The truth table for the JK primitive is shown here:

J	K	Clock	PR*	CL*	Q	Q-BAR
X	X	X	Z,U	X	U	U
X	X	X	X	Z,U	U	U
X	X	X	0	0	1	1
X	X	X	0	1	1	0
X	X	X	1	0	0	1
X	X	X→Z,U	1	1	U	U
X	X	X→0	1	1	ps	ps
0	0	0→1	1	1	ps	ps
0	1	0→1	1	1	0	1
1	0	0→1	1	1	1	0
1	1	0→1	1	1	not Q	not Q-BAR

where X is any value and ps is previous state.

9.6 LATCH PRIMITIVES

There are five latch primitives: the LATCH, LATCH RS, LATCH RS COMP, SCAN LATCH and SCAN LATCH RS. The LATCH, LATCH RS, and LATCH RS COMP primitives have an enable input that is level sensitive. The LATCH RS and LATCH RS COMP also have asynchronous set and reset inputs that cause the outputs to take the values 1 and 0, respectively. The SCAN LATCH and SCAN LATCH RS primitives are used for debugging and testing a design.

LATCH PRIMITIVE

The truth table for the LATCH primitive is:

Data	Enable	Output
X	0	no change
0	1	0
1	1	1
Z,U	1	U
=ps	Z,U	ps
≠ ps	Z,U	U

where X is any value and ps is previous state.

LATCH RS PRIMITIVE

On the LATCH RS primitive, reset prevails over set if both are asserted. The truth table for the LATCH RS primitive is:

Data	Enable	PR*	CL*	Output
X	X	X	Z,U	U
X	X	X	0	0
X	X	Z,U	1	U
X	X	0	1	1
=ps	Z,U	1	1	ps
≠ ps	Z,U	1	1	U
X	0	1	1	ps
0	1	1	1	0
1	1	1	1	1
Z,U	1	1	1	U

LATCH RS COMP

Complementary outputs are provided on the LATCH RS COMP, and both outputs take the value 1 when both set and reset are asserted. The truth table for the LATCH RS

COMP primitive is:

Data	Enable	PR*	CL*	Output	Output*
X	X	X	Z,U	U	U
X	X	Z,U	X	U	U
X	X	0	0	1	1
X	X	0	1	1	0
X	X	1	0	0	1
= ps	Z,U	1	1	ps	ps
≠ ps	Z,U	1	1	U	U
X	0	1	1	ps	ps
0	1	1	1	0	1
1	1	1	1	1	0
Z,U	1	1	1	U	U

SCAN LATCH

The SCAN LATCH primitive is a special purpose primitive. Library developers requiring this primitive should contact Valid.

SCAN LATCH RS

The SCAN LATCH RS primitive is a special purpose primitive. Library developers requiring this primitive should contact Valid.

9.7 REGISTER PRIMITIVES

There are five register primitives: the REG, REG RS, REG RS COMP, REG RS COMP 2 and REG CKE. The registers have an edge-sensitive clock input. When the clock input is not bubbled, the primitive's outputs trigger on a positive edge; when the clock input is bubbled, the outputs trigger on a negative edge. The REG RS, REG RS COMP,

and REG RS COMP 2 also have asynchronous set and reset inputs that cause the outputs to take the values 1 and 0, respectively.

REG PRIMITIVE

The truth table for the REG primitive is:

Data	Clock	Output
0	0→1	0
1	0→1	1
Z,U	0→1	U
X	1→0	ps
=ps	X→Z,U	ps
≠ ps	X→Z,U	U

REG RS PRIMITIVE

In the REG RS primitive, reset prevails over set if both are asserted. The truth table for the REG RS primitive is:

Data	Clock	PR*	CL*	Output
X	X	X	Z,U	U
X	X	X	0	0
X	X	Z,U	1	U
X	X	0	1	1
=ps	X→Z,U	1	1	ps
≠ ps	X→Z,U	1	1	U
0	0→1	1	1	0
1	0→1	1	1	1
Z,U	0→1	1	1	U
X	1→0	1	1	ps

REG RS COMP PRIMITIVE

Complementary outputs are provided on the REG RS COMP, and both outputs take the value 1 when both set and reset are asserted. The truth table for the REG RS COMP primitive is:

Data	Clock	PR*	CL*	Output	Output*
X	X	X	Z,U	U	U
X	X	Z,U	X	U	U
X	X	0	0	1	1
X	X	0	1	1	0
X	X	1	0	0	1
=ps	X→Z,U	1	1	ps	ps
≠ ps	X→Z,U	1	1	U	U
X	X→0	1	1	ps	ps
0	0→1	1	1	0	1
1	0→1	1	1	1	0
Z,U	0→1	1	1	U	U

REG RS COMP 2 PRIMITIVE

This primitive is a special purpose version of the REG RS COMP primitive. The primitive works the same as the REG RS COMP primitive except that both outputs take the value 0 when both PR* and CL* are asserted and several additional conditions govern the behavior of preset and clear. The truth table for the REG RS COMP 2 primitive is:

Data	Clock	PR*	CL*	Output	Output*
X	X	X	Z,U	U	U
X	X	Z,U	X	U	U
X	X	0	0	0	0
X	X	0	1	1	0
X	X	1	0	0	1
=ps	X→Z,U	1	1	ps	ps
≠ ps	X→Z,U	1	1	U	U
X	X→0	1	1	ps	ps
0	0→1	1	1	0	1
1	0→1	1	1	1	0
Z,U	0→1	1	1	U	U

The following additional conditions override the values in the truth table.

1. When an instance of the REG RS COMP 2 primitive has the body property DELAY attached with a value *d*, and when PR* and CL* both change value from 0 to 1 within *d* nanoseconds or less of each other, then both OUTPUT and OUTPUT* take the value U (unknown).

2. When PR* = U and LAST OUTPUT = 1, then OUTPUT = 1, and OUTPUT* = 0.

   When PR* = U and LAST OUTPUT not = 1, then OUTPUT = U, and OUTPUT* = U.

When CL* = U and LAST OUTPUT = 0, then OUTPUT = 0, and OUTPUT* = 1.

When CL* = U and LAST OUTPUT not = 0, then OUTPUT = U, and OUTPUT* = 0.

When PR* = U and CL* = 0, then OUTPUT = 0, and OUTPUT* = U.

REG CKE PRIMITIVE

The REG CKE primitive is similar to the REG primitive except that it has a clock enable input that enables the clock when asserted.

9.8 MULTIPLEXER PRIMITIVES

There are three multiplexer primitives with 2, 4, and 8 inputs: the 2 MUX, 4 MUX, and 8 MUX respectively. The SELECT inputs for these parts have a fixed width of 1, 2, and 3 bits respectively. Use of a multiplexer can often dramatically reduce the number of Simulator primitives needed to model a part. The truth table for the 2 MUX appears below. The table can be extended readily for the 4 MUX and 8 MUX:

S	I0 : I1	Y
0	I0 ≠ Z	I0
	I0 = Z	U
1	I1 ≠ Z	I1
	I1 = Z	U
Z,U	I0 = I1 ≠ Z	I1
	I0 = I1 = Z	U
	I0 ≠ I1	U

9.9 MEMORY PRIMITIVE

There is one memory primitive: MEMORY. The width of each word is determined by the SIZE property. The number of words is determined by the DEPTH property. The ADR input has a size corresponding to the number of words. For example, a 256 word RAM has an ADR input of width eight. As a convenience to the model builder, the WE (write enable) and CS (chip select) inputs on the MEMORY primitive are bubbled because many actual memory parts have these inputs low asserted. These pins may be un-bubbled (use the BUBBLE command in GED) if necessary. The MR (master reset) input, when asserted, clears the entire MEMORY to zeros.

Memories may be modeled in either two-state or four-state mode. Use the MEM_STATE directive to select two-state mode. The default is four-state mode. In two-state mode, each bit of the memory may assume one of two states: 0 and 1. In four-state mode, each bit of the memory may assume one of three states: 0, 1 or U.

Truth tables for each mode are given below. The OUTPUT column shows what value is output in each case. LOC means that the addressed location is output. The WRITE column indicates whether a write operation is performed. "No" indicates that no write operation is performed. A single letter indicates a write operation to the addressed location, and the value written. "All" indicates that the given value is written to all memory locations.

FOUR STATE MODE					
MR	CS	WE	ADR	OUTPUT	WRITE
0	0	X	X	Z	no
0	1,U	X	>= DEPTH	U	no
0	1	0	UNDEF	U	no
0	1	0	DEF	LOC	no
0	1	1	DEF	I	I
0	1	U	DEF	U	U
0	1	1,U	UNDEF	U	U all
0	U	0	X	U	no
0	U	1,U	DEF	U	U
0	U	1,U	UNDEF	U	U all
1	0	X	X	Z	0 all
1	1	X	X	0	0 all
1	U	X	X	U	0 all
U	0	X	X	Z	U all
U	1	X	X	U	U all
U	U	X	X	U	U all

TWO STATE MODE					
MR	CS	WE	ADR	OUTPUT	WRITE
0	0	X	X	Z	no
0	1,U	X	>= DEPTH	U	no
0	1	0	UNDEF	U	no
0	1	0	DEF	LOC	no
0	1	1	DEF	I	I
0	1	U	DEF	1	1
0	1	1,U	UNDEF	1	1 all
0	U	0	X	U	no
0	U	1,U	DEF	U	1
0	U	1,U	UNDEF	U	1 all
1	0	X	X	Z	0 all
1	1	X	X	0	0 all
1	U	X	X	U	0 all
U	0	X	X	Z	1 all
U	1	X	X	1	1 all
U	U	X	X	U	1 all

9.10 COUNTER/SHIFT REGISTER PRIMITIVE

The COUNTER SHIFT REGISTER primitive operates as either a modulo-16 up/down counter or as a 4-bit bidirectional shift register. This primitive has seven inputs:

MR	Master reset
CK	Clock
CEP	Count enable parallel input (active low)

CET — Count enable trickle input (active low). This input also acts as a serial input for shift left.

S — select inputs (3 bits)

DI — parallel data in

MSBIN — serial data input for shift right. This input produces two outputs: DO - data out and TC - terminal count (active low).

The function of the COUNTER SHIFT REGISTER primitive is selected based on the S input as follows:

S2	S1	S0	FUNCTION
L	L	L	Parallel Load
L	L	H	Complement
L	H	L	Shift Right
L	H	H	Shift Left
H	L	L	Count Down
H	L	H	Clear
H	H	L	Count Up
H	H	H	Hold

The output also can be cleared asynchronously by bringing the master reset signal active.

The two count enable inputs are provided for ease of cascading in multistage counters. These two enable inputs must be both asserted for the count up/down operations. One count enable (CET) input also serves as a data input for the shift left operation. The output also can be cleared asynchronously by bringing the master reset signal active.

9.11 ARITHMETIC PRIMITIVES

There are five arithmetic primitives: ADDER, ALU, LOOKAHEAD, CARRY SAVE ADDER, and COMPARATOR.

ADDER PRIMITIVE

The ADDER primitive takes three inputs: A, B, and CARRY IN; and produces four outputs: F, P, G, and CARRY OUT. The size property determines the width of A, B, and F. F takes the sum of A, B, and CARRY IN. CARRY OUT is asserted if an overflow occurs. G is asserted if the addition of A and B generates a carry. P is asserted if the addition of A, B, and 1 propagates a carry.

ALU PRIMITIVE

The ALU primitive has inputs and output identical to those of the adder primitive with the addition of a 4-bit select input that selects a function from the following table:

SELECT	FUNCTION
0	A plus B (BCD)
1	A minus B (BCD)
2	B minus A (BCD)
3	0 minus B (BCD)
4	A plus B
5	A minus B
6	B minus A
7	0 minus B
8	(A and B) or (-A and -B)
9	(A and -B) or (-A and B)
10	A or B
11	A
12	-B
13	B
14	A and B
15	0

BCD stands for binary-coded-decimal. The behavior of the BCD functions is not defined for SIZE values that are not multiples of four, or for data inputs that are not valid BCD values. The "plus" and "minus" denote two's-complement arithmetic. A "-" denotes one's-complement. The ALU primitive is patterned after the 100181 ECL part.

LOOKAHEAD PRIMITIVE

The LOOKAHEAD primitive is a look ahead carry generator with three inputs: P, G, and CARRY IN. The primitive produces one output CARRY OUT. CARRY IN is one bit wide. P, G, and CARRY OUT can be sized. Each CARRY OUT bit is the carry calculated from CARRY IN and the P and G inputs from the least significant bit through the CARRY OUT bit of the primitive.

CARRY SAVE ADDER PRIMITIVE

The CARRY SAVE ADDER takes three inputs: A, B, and C, and produces two outputs: T and CARRY. All can be sized. The 2-bit sum is computed for each bit of A, B, and C and is stored in the corresponding bits of CARRY and F. F is the low order bit of the sum, and CARRY is the high order bit of the sum.

COMPARATOR PRIMITIVE

The COMPARATOR primitive takes two inputs A and B and produces three 1-bit outputs: LT, EQ, and GT. LT is asserted if A < B. EQ is asserted if A = B. GT is asserted if A > B.

9.12 TIMING CHECKER PRIMITIVES

Four timing checker primitives identical to the ones supported in the Timing Verifier are available as simulation primitives. These are: SETUP HOLD, SETUP RISE HOLD FALL, EDGE TO EDGE, and MIN PULSE WIDTH.

Simulation speed is slower when these primitives are used. The TIMING_CHECK directive is used to enable and disable the timing checker primitives.

SETUP HOLD

The SETUP HOLD primitive has a clock and data input. For an active-high clock, it generates an error message in the output listing when the data input is not stable from SETUP ns before the beginning of the rising edge of the clock until HOLD ns after the clock is high. The SETUP HOLD primitive has by default two body properties attached:

SETUP = 0.0
HOLD = 0.0

The properties SETUP and HOLD are assigned the required property values by using the CHANGE command. This primitive is used to check the set-up and hold times of registers and latches.

The SETUP HOLD primitive has an optional enable input, which if specified, turns the checking on and off. If the enable input is any value other than ZERO, then checking is enabled. If checking is enabled any time during the rising edge of the clock input, then checking is performed for that edge.

SETUP RISE HOLD FALL

The SETUP RISE HOLD FALL primitive has a clock and data input. For an active-high clock, it generates an error message in the output listing when the data input is not stable from SETUP ns before the beginning of the rising edge of the clock, while the clock is rising, while the clock is high, during the falling edge of the clock, until HOLD ns after the clock has gone low. The SETUP RISE HOLD FALL primitive has by default two body properties attached:

SETUP = 0.0
HOLD = 0.0

The properties SETUP and HOLD are assigned the required property values by using the CHANGE command. This primitive is used to check the set-up and hold times of data being written into memories.

The primitive has an optional enable input that can be used to turn off checking. If the enable input is given, then any value other than ZERO will cause checking to be enabled. If checking is enabled at any time between the beginning of the rising edge until the end of the falling edge, checking is performed for that clock pulse.

EDGE TO EDGE

The EDGE TO EDGE primitive has two inputs, CK1 and CK2. It checks that the beginning of a RISING edge on CK2 is at least a minimum delay from the end of a RISING edge on CK1 and that the end of a RISING edge on CK2 is no more than a maximum delay from the beginning of a RISING edge on CK1. The EDGE TO EDGE primitive has by default two body properties attached:

MIN = 0.0
MAX = 0.0

The properties MIN and MAX are assigned the required property values by using the CHANGE command. Only rising delays are used.

If there is no edge on CK2 (that is, if CK2 does not change state), then no error message is generated.

The primitive has an optional enable input, which if specified, turns the checking on and off. If the enable input is any value other than ZERO, then checking is enabled. If checking is enabled any time during the rising edge of CK1, then checking is performed for that edge.

MIN PULSE WIDTH

The MIN PULSE WIDTH primitive has one data input. It checks that its data input has no pulses on it that are low for less than LOW ns, and no pulses on it that are high for less than HIGH ns. The MIN PULSE WIDTH primitive has by default two body properties attached:

LOW = 0.0
HIGH = 0.0

The properties LOW and HIGH are assigned the required property values by using the CHANGE command.

The primitive has an optional enable input, which if specified, turns the checking on and off. If the enable input is any value other than ZERO, then checking is enabled. If checking is enabled any time during a given pulse, then the

width of that pulse is checked.

9.13 ENCODER AND DECODER PRIMITIVES

There are four encoder/decoder primitives: 8-BIT PRIO ENCODER, PRIORITY ENCODER, 1 OF 8 DECODER, and 8-BIT DECODER.

8-BIT PRIO ENCODER PRIMITIVE

The 8-BIT PRIO ENCODER primitive takes an 8-bit input and produces two outputs: T, which is three bits wide and ANY, which is one bit wide. ANY is asserted if any input bit is asserted. T is the bit number of the most significant bit asserted, if any, where 0 is the most significant input.

PRIORITY ENCODER PRIMITIVE

The PRIORITY ENCODER primitive takes eight 1-bit inputs: I7 . . I0, and produces two outputs: T, which is three bits wide, and ANY, which is one bit wide. ANY is asserted if any input bit is asserted. T is the bit number of the most significant input which is asserted, if any, where I7 is the most significant input and has a bit number of 7 (111 binary).

1 OF 8 DECODER PRIMITIVE

The 1 OF 8 DECODER primitive takes two inputs: SELECT, which is three bits wide and ENABLE, which is one bit wide. It produces an 8-bit output T. If ENABLE is asserted, SELECT selects which bit of T is asserted. When SELECT contains Z and/or U, those SELECT bits are treated as "don't care" for selecting output bits and the selected output bits are set to U.

S<2..0>	EN	T<7..0>
X	1	all bits 0
i=defined value	Z,U	the i-th bit U and the rest 0
(0/1,Z/U,0/1) etc	Z,U	the (0/1,*,0/1)-th bits U and the rest 0
i=defined value	0	the i-th bit 1 and the rest 0
(0/1,Z/U,0/1) etc	0	the (0/1,*,0/1)-th bits U and the rest 0

8-BIT DECODER

The 8-BIT DECODER primitive is identical in operation to the 1 OF 8 DECODER primitive. The output of the 1 OF 8 DECODER is an 8 bit bus. The output of the 8-BIT DECODER is eight individual bits.

9.14 OTHER PRIMITIVES

The remaining four Simulator primitives are: PARITY, RES, PASS TRANSISTOR, and UNI PASS TRANSISTOR.

PARITY PRIMITIVE

The PARITY primitive's I input can be sized and produces a one bit output T. T is the odd parity of I.

RES PRIMITIVE

The resistor primitive RES is fully bidirectional and acts like a wire except that HARD strength signals are converted to SOFT strength when they pass through. RES primitives always have 0 delay. The RES primitive is SIZE wide, and the pins may not be bubbled.

PASS TRANSISTOR PRIMITIVE

The PASS TRANSISTOR primitive is fully bidirectional, and acts like a switch. The G pin of the PASS TRANSISTOR controls whether the A and B pins are connected together. An active G pin (0 if the pin is bubbled, otherwise 1) causes the PASS TRANSISTOR to act like a wire, connecting the A and B nets. An inactive G pin causes the PASS TRANSISTOR to act as if it were not in the circuit. The delay from A to B or B to A is always 0. The G pin has an input delay that assumes the value of the DELAY property on the PASS TRANSISTOR. The A and B pins of the PASS TRANSISTOR are SIZE wide and may not be bubbled. The G pin is always one bit wide, and may be bubbled.

UNI PASS TRANSISTOR PRIMITIVE

The UNI PASS TRANSISTOR is a unidirectional version of the PASS TRANSISTOR and results in more rapid simulation for MOS circuits. Pins and properties of the UNI PASS TRANSISTOR primitive are identical - a G pin that controls whether the A and B pins are connected. However, because this transistor is uni-directional, the A pin is an input pin rather than an output.

9.15 USER-CODED PRIMITIVES

The Simulator allows users to code simulator models in PASCAL and refer to them using standard SCALD drawings. Existence of these user-coded primitives (UCPs) means that the user can expand the "parts set" understood by the Logic Simulator. This feature is described in detail in the section "User-Coded Primitives."

9.16 PROPERTIES AFFECTING SIMULATION

When a signal is driven by more than one output, the result depends on the logic family. Output pins are given output types to specify their behavior when wired together. The OUTPUT_TYPE pin property is put on the body

drawings for the part and is inherited by both Simulator and Timing Verifier models. Supported output_type values are:

TS	tri-state
TS,TS	tri-state
OC,AND	open collector
OE,OR	open emitter

If no output type is given, the pin behaves as a "totem pole" TTL output.

SECTION 10
ERROR MESSAGES

Error messages are short statements that identify and record each error encountered by the Logic Simulator. When errors occur in a run of the Simulator, error messages are generated and printed in the list file (simlst.dat) in numerical sequence.

10.1 NUMERICAL LISTING OF ERROR MESSAGES

The rest of this section contains a numerical listing of all of the Logic Simulator error messages. Each message is explained and often suggestions are made about the causes of the problem and how to remedy them.

Certain error messages are labeled "reserved". This means that the error does not occur in normal operation and is reserved by Valid for debugging or other internal operations.

ERROR #1: Expected identifier

Generated when the Simulator is expecting an identifier (a string of letters, digits, or '__' starting with a letter) and finds some other data.

ERROR #2: Expected =

Generated when the Simulator is expecting an equal sign (=) and finds some other data.

ERROR #3: Expected [

Generated when the Simulator is expecting a left square bracket [and finds some other data.

ERROR #4: Expected]

Generated when the Simulator is expecting a right square bracket] and finds some other data.

ERROR #5: Expected a constant

Generated when the Simulator is expecting a constant and finds some other data.

ERROR #6: Expected (

Generated when the Simulator is expecting a left parenthesis (and finds some other data.

ERROR #7: Expected)

Generated when the Simulator is expecting a right parenthesis) and finds some other data.

ERROR #8: Expected ,

Generated when the Simulator is expecting a comma and finds some other data.

ERROR #9: Expected *

Generated when the Simulator is expecting an asterisk * and finds some other data.

ERROR #10: Expected <

Generated when the Simulator is expecting a less than character < and finds some other data.

ERROR #11: Expected >

Generated when the Simulator is expecting a greater than character > and finds some other data.

ERROR #12: Expected ;

Generated when the Simulator is expecting a semicolon ; and finds some other data.

ERROR #13: Expected :

Generated when the Simulator is expecting a colon : and finds some other data.

ERROR #14: Unexpected symbol in integer expression

Generated when the Simulator is reading an expression and finds something unexpected. When this error occurs, the Simulator is expecting one of the following:

1. A constant
2. An expression in parentheses, for example, (2+3).
3. NOT followed by an item from this list.
4. An identifier whose value is one of the above or a parameter whose value is an integer.

ERROR #15: Reserved

ERROR #16: Bit value invalid

Generated when the Simulator is reading a bit subscript and finds an illegal bit value. Bit values are invalid if they are negative or are greater than the largest allowed bit number. Since the largest allowed bit number is (2**31 - 1 = 2147483647), this error usually means that the bit value

is negative.

ERROR #17: Expected /

Generated when the Simulator is expecting a slash / and finds some other data.

ERROR #18: Scald configuration file does not exist

Generated when the Simulator is unable to find the SCALD configuration file (config.dat). Check the location and protection of the file.

ERROR #19: Reserved

ERROR #20: Unmatched closing comment symbol

Generated when the Simulator encounters a closing comment symbol } (right curly brace) without a matching starting comment symbol { (left curly brace). Either this symbol is extraneous or the beginning of the comment was never specified.

ERROR #21: Reserved

ERROR #22: String length exceeded

Generated when the Simulator is reading a string and finds that the string is too long. Strings are limited to 255 characters. The Simulator ignores the characters from the current position to the end of the string.

ERROR #23: Illegal character found

Generated when the Simulator finds an illegal character in the input line. Removing the character will solve the problem.

ERROR #24: Expression value overflow

Generated when the Simulator evaluates an expression and its value overflows. When this error occurs, the Simulator assigns 0 to the expression and continues.

ERROR #25: Division by zero

Generated when the Simulator detects a division by 0 during the evaluation of an expression. This error does not abort the Simulator, but the division is skipped.

ERROR #26: Could not parse signal name.

Generated when the Simulator fails to parse a signal name. This error is usually caused by a typing error. Check the signal name and enter again.

ERROR #27: Reserved

ERROR #28: Special sampling not supported in RM

Generated when the Simulator finds a Realmodel device using the "SAMPLE=SPECIAL" directive in its definition file. Delete this directive from the definition file or plug the device into Realchip instead of Realmodel.

ERROR #29: Expected SPECIAL

Generated when using Realchip or Realmodel. The Simulator expected SPECIAL keyword and found some other data in the definition file.

ERROR #30: Unexpected symbol in bit subrange

Generated when the Simulator finds an unexpected symbol in a bit subrange. The symbols expected by the Simulator in a bit subscript are:

1. Two dots specifying a subrange (..).

2. A colon (:) specifying a bit step.

3. A greater than symbol (>).

ERROR #31: Realfast not supported on this platform.

Generated when the IBM PC platform is being used and the Simulator finds the USE_REALFAST ON directive. Realfast cannot be used with the IBM PC platform. Remove the directive.

ERROR #32: Non-printing ASCII character found

Generated when the Simulator finds a non-printing ASCII character in the input line. Deleting the character corrects this error.

ERROR #33: Expected a string

Generated when the Simulator is expecting a string and finds some other data.

ERROR #34: Comment not closed before end of input

Generated when the Simulator does not find the end of the comment before the end of input.

ERROR #35: Cannot create instance in Realfast

Generated when not enough Realfast data structure memory is available to store the information for a Realchip instance found in the drawing. Run on a Realfast with more (event) memory.

ERROR #36: Reserved

ERROR #37: Expected .

Generated when the Simulator is expecting a period . and finds some other data.

ERROR #38: Illegal device size

Generated when the Simulator finds the number_dev_pins specified in the Realchip definition file is not a multiple of 64 or is greater than 1536.

ERROR #39: Undefined identifier in expression

Generated when the Simulator finds an undefined identifier in the expression. Identifiers are used as names in properties. Check the DEFINE bodies and parameters of the body.

ERROR #40: Expected END

Generated when the Simulator is expecting the keyword 'END' and finds some other data. Check the file for the keyword at the end of the file.

ERROR #41: Identifier length exceeded

Generated when the Simulator encounters an identifier that has more than 16 characters. The Simulator ignores the rest of the characters in the identifier.

ERROR #42: Non-existent primitive in expansion file

Generated when the Simulator encounters a primitive in the expansion file that is not a Simulator, user-coded, or Realchip primitive.

ERROR #43: Non-existent pin on primitive

Generated when the Simulator finds a pin on a primitive in the expansion file that is not a defined pin on the primitive.

ERROR #44: Illegal output type

Generated when an improper output type is detected for the OUTPUT_TYPE pin property.

ERROR #45: Pin can have only one OUTPUT_TYPE

Generated when the Simulator encounters more than one output type for a pin. Defining exactly one output type corrects this error.

ERROR #46: Failed to add ptrn to Realfast

Generated when not enough Realfast data structure memory is available to store a new Realchip pattern. Run on a Realfast with more (event) memory.

ERROR #47: RM hardware failure

Generated when some unexpected Realmodel hardware failure occurs. An interrupt was received but the hardware was still running when the interrupt was serviced. Please contact Valid.

ERROR #48: Command file already specified-ignoring

Generated when the Simulator encounters more than one COMMAND_FILE directive. Only one command file can be specified in the directives file; all command files except the first are ignored.

ERROR #49: Cannot specify both types of tracing

Generated when the user specifies both BINARY_TRACEing and TABULAR_TRACEing in the directives file. The BINARY_TRACE directive is ignored.

ERROR #50: Reserved

ERROR #51: Unknown directive

Generated when the Simulator encounters an unknown directive in the directives file.

ERROR #52: Invalid specification for directive

Generated when the Simulator is processing a directive from the directives file and encounters an invalid operand.

ERROR #53: Input line exceeds maximum length

Generated when the Simulator tries to read a line greater than 255 characters. The input line must be divided to correct this error.

ERROR #54: Expected error number for suppression

Generated when the Simulator is expecting an error number to be suppressed and finds some other data.

ERROR #55: This error cannot be suppressed

Generated when the user tries to suppress an error that cannot be suppressed (only errors classified as oversights or warnings can be suppressed).

ERROR #56: Realmodel parity error

Generated when the Simulator finds a hardware error (parity error) in either the Realmodel Interface Board, a Realmodel Slave Board, or the Realmodel Master Board. Please contact Valid.

ERROR #57: End of input before end of expression

Generated when the Simulator finds the end of input before the end of the expression being evaluated.

ERROR #58: Illegal dev_type

Generated when the Simulator finds an illegal dev_type in Realchip. Realchip does not support devices with a size greater than 128 pins.

ERROR #59: Realmodel hardware never stopped

Generated when the Simulator finds that the stop bit in the Realmodel hardware status register is not set by the time the Simulator is ready to play the environment sequence. This can indicate either a hardware or software failure. Please contact Valid.

ERROR #60: Number of errors must be > 0

Generated when the Simulator is processing the MAXIMUM_ERRORS directive and finds a 0 or negative number.

ERROR #61: Radix must be in the range 2..16

Generated when the Simulator encounters a radix (base) outside the range 2-16. The Simulator supports four radices: 2, 8, 10, and 16.

ERROR #62: Trace radix must be 2,b,8,o,10,d,16 or h

Generated when the Simulator finds a specification for the radix in a TRACE file other than the indicated values.

ERROR #63: Cannot open command log file (SIMCMD)

Generated when the Simulator is unable to open the command log file for output. Check disk space and directory protection.

ERROR #64: Cannot open Simulator list file (SIMLST)

Generated when the Simulator is unable to open the list file for output. Check disk space and directory protection.

ERROR #65: Cannot open session log file (SIMLOG)

Generated when the Simulator is unable to open the session log file for output. Check disk space and directory protection.

ERROR #66: Cannot open error log file (OUTFILE)

Generated when the Simulator is unable to open the error log file for output. Check disk space and directory protection.

ERROR #67: Incorrect envir. vars or terminal type

Generated when the Simulator cannot find the environment variables (TERMCAP) for the terminal, or finds that a terminal is set to GCLUSTER when it isn't one. Set the proper terminal type or correct the environment variables.

ERROR #68: Min. graphics Sim. window at least 22x80

Generated when the graphics Simulator is invoked in a window which is smaller than the minimum size of 22 x 80. Create a larger window for the Simulator.

ERROR #69: Reserved

ERROR #70: Non-contiguous bit subscripts for pin

Generated when the Simulator finds non-contiguous bit subscripts for a pin on a primitive. The Simulator supports only contiguous bit subscripts.

ERROR #71: Window too small-must be at least 12x80

Generated when the Simulator is invoked in a window smaller than 12 x 80. Create a larger window for the Simulator.

ERROR #72: Unknown signal syntax specification

Generated when the Simulator finds a syntax specification with which it is not familiar. Check the syntax specification.

ERROR #73: Signal syntax element found twice

Generated when the Simulator finds an element specified twice while reading the signal syntax specification. Removing the second specification corrects this error.

ERROR #74: Every syntax MUST have a name portion

Generated when the Simulator does not find a name portion in the signal syntax specification.

ERROR #75: Every syntax MUST have a subscript

Generated when the Simulator does not find a subscript portion in the signal syntax specification.

ERROR #76: Illegal form for signal syntax

Generated when the Simulator finds an element that is not assertion_specifier, negation_specifier, or name_specifier while reading the signal syntax, or does not have the name_specifier in the proper location.

ERROR #77: Symbol must be one character

Generated when the Simulator finds more than one character as the assertion_specifier symbol.

ERROR #78: Illegal symbol as configuration char

Generated when the Simulator finds a forbidden symbol (; < > # 0-9) as the configuration character.

ERROR #79: Error in remote host simulation

Generated when the Simulator gets illegal data from the Network Realchip Server. Please contact Valid.

ERROR #80: Error in remote sendnet

Generated when the Simulator detects an error while sending data over the network. This may be due to the Network Realchip Server being down or to a problem with the Ethernet cable connection.

ERROR #81: Error in remote readnet

Generated when the Simulator detects an error while reading data over the network. This may be due to the Network Realchip Server being down or to a problem with the

Ethernet cable connection.

ERROR #82: Root drawing has some compile errors

Generated when there are errors reported by the Compiler when it is invoked from the Simulator. Correct the compile errors.

ERROR #83: Root drawing does not exist

Generated when the Simulator is unable to find the drawing specified by the ROOT_DRAWING directive. Check the drawing name.

ERROR #84: Cannot open WIREDELAYS file

Generated when the Simulator is unable to open the wire delay file for reading.

ERROR #85: Expected FILE_TYPE specification

Generated when the Simulator does not find a file_type specification in the file it is currently reading.

ERROR #86: File is not of the correct type

Generated when the Simulator finds a file_type that is not the correct type for the current file.

ERROR #87: Directory file name previously specified

Generated when the Simulator finds that a SCALD directory has been specified more than once in the directives file. Removing the second entry in the directives file corrects this error.

ERROR #88: Cannot open tabular trace output file

Generated when the Simulator is unable to open an output file for writing tabular trace.

ERROR #89: String not closed before the end of line

Generated when the Simulator finds that a string does not have a closing quote before the end of the line.

ERROR #90: Cannot access device on remote host

Generated when the Simulator is unable to access a Realchip device on a remote host. The device may not be present on the remote host or the device on the remote host may not be available (static_forever device is being used by another user).

ERROR #91: Unknown service: check /etc/services

Generated when the Simulator is unable to communicate with a remote host running the Network Realchip Server. The /etc/services file should contain the entry "realchip 900/tcp".

ERROR #92: Unknown remote host or remote host down

Generated when the Simulator is unable to access the remote host specified in the REMOTE_HOST directive. The host may not be on the network, the host may be down, or the name of the remote host may not be defined in /etc/hosts.

ERROR #93: Server busy or down on the remote host

Generated when the Simulator is unable to access a Network Realchip Server. The server may be busy or may not be running on the remote host.

ERROR #94: Cannot open signal mapping file(SIGMAP)

Generated when the Simulator is unable to open the signal mapping file.

ERROR #95: Cannot open synonym file

Generated when the Simulator is unable to open the synonyms file for reading. Misspelled or improper specification of the synonyms file pathname in the directives file can cause this error.

ERROR #96: Cannot open MEMLOAD file

Generated when the Simulator is unable to open the file specified in the MEMLOAD command.

ERROR #97: Expansion file not for Simulator

Generated when the Simulator finds an incorrect file type in the Compiler expansion file. The drawing was not compiled for sim.

ERROR #98: This property has already been specified

Generated when the Simulator finds a property of a body defined more than once.

ERROR #99: Error limit exceeded

Generated when the Simulator detects more than the maximum number of errors.

ERROR #100: Assertion chk failure: save simlog file

Generated when an internal error (an assertion failure) is detected by the Simulator. Please contact Valid.

ERROR #101: Cannot open compiler output (CMPEXP)

Generated when the Simulator is unable to open the Compiler expansion file. Check the directives file and verify the pathname.

ERROR #102: Compiler synonyms file has wrong type

Generated when the Simulator finds an incorrect file type for the synonyms file. Recompile the drawing for sim.

ERROR #103: Cannot open user input file (USERIN)

Generated when the Simulator is unable to start terminal interaction, in interactive mode, or the script file, in batch mode.

ERROR #104: Unknown command

Generated when the Simulator does not recognize the command entered. Check the manual for the proper Simulator commands.

ERROR #105: Malformed command

Generated when a non-identifier is entered as a command to the Simulator.

ERROR #106: Cannot open tabular trace input(TABULAR)

Generated when the Simulator is unable to open the trace file for reading.

ERROR #107: Cannot open directives file (INFILE)

Generated when the Simulator is unable to find the simulate.cmd file in the default or specified directory.

ERROR #108: Clock signal must be undriven

Generated when the Simulator finds a clock signal that is not undriven -- building a clock on a driven signal results in unexpected behavior.

ERROR #109: Clock time must be within clock period

Generated when the Simulator finds a transition time that is greater than the clock period while building the clock transitions list. The Simulator assigns the transition time to be the clock period and builds the list accordingly.

ERROR #110: Clock time less than previous time

Generated when the Simulator finds a transition time that is less than the previous transition time while building the clock transitions list. The signal should have ascending clock assertions to correct this error.

ERROR #111: Clock period must be greater than 0

Generated when the Simulator finds a negative or zero clock period in the PERIOD command or CLOCK_PERIOD directive. Specify a clock period greater than zero.

ERROR #112: Run stopped because errors were detected

Generated when the Simulator halts after encountering fatal errors. Check previous errors detected by the Simulator.

ERROR #113: Clock intervals must be greater than 0

Generated when the Simulator finds a value less than 1 for the number of clock intervals. Specify a value greater than zero.

ERROR #114: Only 2 or 4 memory states are allowed

Generated when the Simulator finds an operand other than 2 or 4 while processing the MEM_STATE directive.

ERROR #115: Illegal parameter to OUTPUT directive

Generated when the Simulator finds some parameter other than list or command_log to the OUTPUT directive.

ERROR #116: Illegal terminal type

Generated when the Simulator finds an improper terminal type while processing the TERMINAL command or TERMINAL directive. See the Directives section for allowed terminal types.

ERROR #117: Illegal value for memory depth

Generated when the Simulator finds a value that is negative, zero, or greater than the maximum depth for the memory primitive.

ERROR #118: Expected BIT_RANGE

Generated when the Simulator is expecting a bit range and finds some other data.

ERROR #119: Expected ..

Generated when the Simulator is expecting '..' and finds some other data.

ERROR #120: Expected MEM_BLOCK

Generated when the Simulator is expecting a MEM_BLOCK and finds some other data. Check the MEMLOAD file.

ERROR #121: Input word is wider than memory

Generated when the Simulator reads a word from a MEMLOAD file and finds that it is larger than the memory word. Either adjust the input to the proper width or use the subrange specification of the MEMLOAD command.

ERROR #122: Expected END_MEM_BLOCK

Generated when the Simulator is expecting an END_MEM_BLOCK symbol and finds some other data. Check the MEMLOAD file.

ERROR #123: BIT_RANGE does not match memory width

Generated when the Simulator reads a MEMLOAD file and finds that the bit range specified in the file does not match the memory width. Use or correct the bit range specification option of the MEMLOAD command.

ERROR #124: Illegal BIT_RANGE bit ordering

Generated when the Simulator finds a reversed bit range specification (right_to_left ordering when using left_to_right ordering). Reversing the bit range corrects this error.

ERROR #125: Reserved

ERROR #126: Memory contents file has wrong type

Generated when the Simulator is reading a MEMLOAD file and finds a wrong type in the file. Use the correct file_type specification in the file.

ERROR #127: Input word is narrower than memory

Generated when the Simulator is reading a MEMLOAD file and finds that the word read is narrower than the memory word.

ERROR #128: Fewer words than specified in MEM_BLOCK

Generated when the Simulator is expecting more words in a MEMLOAD file and finds an END_MEM_BLOCK symbol.

ERROR #129: Cannot open user configuration file

Generated when the Simulator is unable to open the user-coded primitive configuration file. Check the file pathname in the directives file.

ERROR #130: Expected PRIMITIVE

Generated when the Simulator is expecting a PRIMITIVE symbol and finds some other data.

ERROR #131: Primitive already defined

Generated when the Simulator finds a primitive defined more than once. Remove the extra declaration.

ERROR #132: Expected PIN

Generated when the Simulator is expecting a PIN description and finds some other data.

ERROR #133: Expected INPUT_SPEC or OUTPUT_SPEC

Generated when the Simulator is expecting either the INPUT_SPEC or OUTPUT_SPEC symbol and finds some

other data.

ERROR #134: Expected END_PIN

Generated when the Simulator is expecting an END_PIN symbol and finds some other data.

ERROR #135: Expected END_PRIMITIVE

Generated when the Simulator is expecting an END_PRIMITIVE symbol and finds some other data.

ERROR #136: Expected width specification

Generated when the Simulator is expecting a width specification and finds some other data.

ERROR #137: Illegal OWN_STORAGE value

Generated when the Simulator is expecting the number of storage words and finds a value that is negative, zero, or greater than the maximum user storage.

ERROR #138: Cannot open Realchip Library file

Generated when the Simulator is unable to open the Realchip library file. Check the pathname in the directives file.

ERROR #139: Expected INPUT_SPEC,OUTPUT_SPEC, IO_SPEC

Generated when the Simulator is expecting the INPUT_SPEC or OUTPUT_SPEC symbol in a UCP and finds some other data, or one of these two symbols or IO_SPEC in the Realchip definition file and finds some other data.

ERROR #140: Illegal JIG_ID value

Generated when the Simulator finds an invalid JIG_ID for the Realchip primitive used in the simulation.

ERROR #141: Expected DYNAMIC,STATICor STATIC_FOREVER

Generated when the Simulator is expecting the DYNAMIC, STATIC, or STATIC_FOREVER symbols and finds some other data.

ERROR #142: Expected RISE, FALL, or BOTH

Generated when the Simulator is expecting a rise or fall delay or both delays and finds some other data.

ERROR #143: Illegal clock_period value(s)

Generated when the Simulator finds a negative or zero clock period value.

ERROR #144: Expected , or ;

Generated when the Simulator is expecting a comma or semicolon (, or ;) and finds some other data.

ERROR #145: Expected : or , or ;

Generated when the Simulator is expecting a colon, comma, or semicolon and finds some other data.

ERROR #146: Pin number out of range

Generated when the Simulator finds a pin number that is out of range. Since the number of pins supported is very large, this error seldom occurs.

ERROR #147: Delay value out of range

Generated when the Simulator finds a delay value that is either less than the minimum value or greater than the maximum value.

ERROR #148: Expected (TS,TS), (OC,AND) or (OE,OR)

Generated when an OUTPUT_TYPE property is not followed by either a (TS,TS), (OC,AND), or (OE,OR) value.

ERROR #149: Unknown definition parameter

Generated when the Simulator is unable to interpret a parameter of a pin specification.

ERROR #150: Expected END_RESET_SEQ

Generated when the Simulator is expecting an END_RESET_SEQ symbol and finds some other data while reading a Realchip library.

ERROR #151: RESET_SEQ pin not found

Generated when the Simulator does not find the pin specified in the reset sequence block of the Realchip definition file. Check that the pin names match.

ERROR #152: Expected 0, 1 or Z

Generated when the Simulator is expecting a 0, 1 or Z and finds some other data.

ERROR #153: CLOCK_PIN pin not found

Generated when the Simulator finds that the pin specified with the CLOCK_PIN directive has not been defined in the pin section of the Realchip definition file.

ERROR #154: Realchip adapter not found

Generated when the Realchip device is used and the Simulator does not find the adapter.

ERROR #155: Cannot open trace value file (VALBIN)

Generated when the Simulator is unable to open the trace value file for output.

ERROR #156: Cannot open trace value file (VALASC)

Generated when the Simulator is unable to open the trace value file for output.

ERROR #157: Expected END_DELAY_TABLE

Generated when the Simulator does not find the END_DELAY_TABLE symbol while reading the Realchip library.

ERROR #158: DELAY_TABLE output pin not found

Generated when the Simulator does not find the output pin read in the DELAY_TABLE. Check the pin name.

ERROR #159: Trace interval must be 0 or greater

Generated when the Simulator finds a negative trace interval.

ERROR #160: Symbol_stack overflow

Generated when the Simulator symbol stack exceeds its maximum depth during parsing.

ERROR #161: Tabular trace input file has wrong type

Generated when the Simulator finds an incorrect file type while reading the tabular trace input file.

ERROR #162: Expected END_TAB_TRACE

Generated when the Simulator is expecting an END_TAB_TRACE symbol while reading the tabular trace file and finds some other data.

ERROR #163: Expected START_TAB_TRACE

Generated when the Simulator is expecting a START_TAB_TRACE symbol while reading the tabular trace file and finds some other data.

ERROR #164: Stimulation time must be > current time

Generated when the Simulator finds the stimulation time to be less than the current time.

ERROR #165: Incorrect signal value in Tabular file

Generated when the Simulator finds an erroneous signal value in the tabular file.

ERROR #166: Radix must be 2,8,10,16 in Tabular file

Generated when the Simulator finds a value other than 2, 8, 10, or 16 for the radix of a value in the tabular input file.

ERROR #167: Decay time must be greater than 0

Generated when the Simulator finds the decay time to be less than zero.

ERROR #168: Odd # of reset_seq for clk_both device

Generated when the device definition file has an odd number of reset_sequences for the particular device (clock_type = clock_both). Modify the definition file so that the device has an even number of reset_sequences.

ERROR #169: Adapter found with duplicate jig ID

ERROR #170: Pattern RAM overflow. Invalid results.

Generated when the Realchip simulation pattern RAM overflows.

ERROR #171: Unsupported clock period range

Generated when the Simulator finds a clock period that is outside the range supported by Realchip/Realmodel.

ERROR #172: Missing pin number specification

Generated when the Simulator is expecting a pin number specification and finds some other data.

ERROR #173: DELAY_TABLE input pin not found

Generated when the Simulator does not find the input pin read from the DELAY_TABLE. Check the pin specification.

ERROR #174: PAUSE sequence already given

Generated when the Simulator encounters a PAUSE sequence more than once.

ERROR #175: Pin number is already used

Generated when the Simulator encounters a pin number that is already used for another pin.

ERROR #176: Cannot find available modeling system

Generated when a user attempts to use Realchip or Realmodel on a system where it does not exist or is already in use.

WARNING #177: Feature not yet implemented for Realfast

Generated when the Simulator finds that the user tried to invoke a Simulator feature that is not available when using Realfast.

ERROR #178: Feedback may be disconnected

Generated when the feedback connection for a device that requires feedback is disconnected.

ERROR #179: Setup timing violation

Generated when the SETUP HOLD or SETUP RISE HOLD FALL timing checkers encounter a setup time violation. Related signal names and the violation time are printed in the Simulator output files.

ERROR #180: Hold timing violation

Generated when the SETUP HOLD or SETUP RISE HOLD FALL timing checkers encounter a hold time violation. Related signal names and the violation time are printed in the Simulator output files.

ERROR #181: Edge to Edge timing violation

Generated when the EDGE TO EDGE timing checker encounters an edge to edge timing violation. Related signal names and the violation time are printed in the Simulator output files.

ERROR #182: Cannot open memory dump file

Generated when the Simulator is unable to open an output file for dumping the contents of a memory primitive. Check disk space and directory protection.

ERROR #183: Cannot close output file

Generated when the Simulator is unable to close an output file.

ERROR #184: Cannot close input file

Generated when the Simulator is unable to close an input file.

ERROR #185: WIRE_ESTIMATE family has been redefined

Generated when the Simulator finds more than one WIRE_ESTIMATE directive with the same family name. The later definition overrides all previous definitions with the same name.

ERROR #186: Have to use root_drawing for PFG

Generated when the probabilistic fault grading (PFG) feature is specified and the ROOT_DRAWING directive has not been used. PFG does not work when the design is read in through an expansion file.

ERROR #187: Cannot open PFG file

Generated when the Simulator is unable to open the PFG file. The file may not exist or the user may not have read permission.

ERROR #188: Identifier not found,skip the expression

Generated when the Simulator does not find an expression name in the USER_EXPRESSION directive. The expression defined by this directive is ignored.

ERROR #189: PFG is not on

Generated when the user invokes the PFG command during a simulation for which the PFG directive has not been set ON. The PFG feature must be enabled before the command can be invoked.

ERROR #190: PFG strobe period is 0 - turning pfg off

Generated when the PFG_STROBE directive is not specified or is specified as 0 in the directives file. The PFG_STROBE period must be greater than 0.

ERROR #191: Minpulse Width timing violation

Generated when the MIN PULSE WIDTH timing checker encounters a minimum pulse width violation. Related signal names and the violation time are printed in the Simulator output files.

ERROR #192: Too many entries in WIRE_ESTIMATE list

Generated when the WIRE_ESTIMATE directive has more than 100 entries. The value of the entries after the 100th are set to the same value as the 100th entry.

ERROR #193: Maximum WIRE_ESTIMATE entry is 100

Generated when the WIRE_ESTIMATE directive has more than 100 entries. The value of the entries after the 100th are set to the same value as the 100th entry.

ERROR #194: Reserved

ERROR #195: No expression follows condition

Generated when the Simulator finds only a condition in the USER_EXPRESSION directive. The expression is ignored.

ERROR #196: Signame_chars parameter out of range

Generated when the Simulator finds some value in the SIGNAME_CHARS directive which is not in the range 9-24. Check this parameter.

WARNING #197: Improper MEMLOAD params - using defaults

Generated when the Simulator finds some error in the specification of the optional parameters for the MEMLOAD command. Simulator will try to use the default parameters instead.

WARNING #198: Inconsistent MEMLOAD parameters

Generated when the Simulator finds a different number of bits or words in the MEMLOAD file from that specified by the optional parameters for the MEMLOAD command.

WARNING #199: File is longer than memory

Generated when the Simulator finds that the MEMLOAD file has more words than the memory primitive for the MEMLOAD or DUMPMEMORY commands. Use the

word range specification option.

WARNING #200: File has fewer words than memory

Generated when the Simulator finds that the MEMLOAD file has fewer words than the memory primitive being loaded.

ERROR #201: Output already has Realfast data

Generated when the Simulator finds that the data structure for an output already has Realfast data in it.

ERROR #202: Not enough data structure memory

Generated when the Simulator finds that the design is too big to fit in available Realfast memory. In particular, all of the data structure memory (event memory) was used up. Run on a Realfast with more memory.

ERROR #203: Input already has Realfast data

Generated when the Simulator finds that the data structure for an input already has Realfast data in it.

ERROR #204: Primitive not yet implemented

Generated when the Simulator finds that the design contains a simulator primitive which has not been implemented in Realfast. Either change the design to not use that primitive or simulate without using Realfast.

ERROR #205: Not enough microcode memory

Generated when the Simulator finds that the design is too big to fit in available Realfast memory. In particular, all of the evaluation memory was used up. Run on a Realfast with more memory.

ERROR #206: SET_MICRO_FIELD has invalid parameters

Generated when the Simulator finds an illegal value in a field of an instruction in the Realfast microcode. Please contact Valid.

ERROR #207: Output has width > 1

Generated when the Simulator finds an internal error indicating that a Realfast data structure has a width greater than one.

ERROR #208: Undriven input has illegal default value

Generated when the Simulator finds an internal error indicating that a Realfast data structure has an improper default value.

ERROR #209: Reserved

ERROR #210: Primitive already has Realfast data

Generated when the Simulator finds that the data structure for a primitive already has Realfast data in it.

ERROR #211: Monitor code too large

Generated when the Simulator finds that the microcode monitor was larger than expected. This can only happen if the Simulator and /u0/scald/simulator/monitor.int are out of sync. Check installation.

ERROR #212: Monitor returned an error code

Generated when the Simulator finds that the microcode monitor gave a failure indication. This usually indicates a hardware problem but could also indicate an internal consistency failure.

ERROR #213: Did not get access to Realfast hardware

Generated when the Simulator is unable to access the Realfast hardware. This can be caused when there is no Realfast hardware plugged into the S-32, when Realfast is turned off, when the hardware is incorrectly plugged in, or when some other user is using Realfast at the present time.

ERROR #214: Realfast interrupt but hardware busy

Generated when the Simulator finds that Realfast indicated an interrupt condition but was still running when the interrupt was serviced. This probably indicates a hardware failure. If this error occurs, call your field service representative.

ERROR #215: Reserved

ERROR #216: Reserved

ERROR #217: Ran out of event blocks

Generated when the Simulator finds that the Realfast data structure memory was exhausted during simulation. Run on a Realfast with more memory.

ERROR #218: Some delay greater than 4095 detected

ERROR #219: Not enough value memory

Generated when the Simulator finds that the design is too big to fit in available Realfast memory. In particular, all of the evaluation memory was used up. Run on a Realfast with more memory.

ERROR #220: UCP/Realchip delay >= 4096

Generated when the Simulator finds some UCP or Realchip primitive with a delay of 4096 or greater. Change the UCP or Realchip definition file to not use such a large delay.

ERROR #221: Realfast memory parity error

Generated when the Simulator finds a parity error in the Realfast memory during simulation. This indicates a hardware problem; call your field service representative.

ERROR #222: Feature not yet implemented for Realfast

Generated when the Simulator finds that the user tried to invoke a Simulator feature which is not available when using Realfast. This includes logic patching and breakpoints.

ERROR #223: Cannot open Realfast monitor file

Generated when the Simulator finds that it cannot access /u0/scald/simulator/monitor.int. Check that this file is present and readable by users. Also check /usr/bin/simassign to ensure that there is an entry "RFMON=/u0/scald/simulator/monitor.int".

ERROR #224: Cannot open Realfast ALU file

Generated when the Simulator finds that it cannot access /u0/scald/simulator/alumem.int. Check that this file is present and readable by users. Also check /usr/bin/simassign to ensure that there is an entry "RFALU=/u0/scald/simulator/alumem.int".

ERRORS #225 through #239: Reserved

ERROR #240: Improper operands order

Generated when the Simulator finds that the operands in the USER_EXPRESSION directive are in the wrong order. Output in the list file indicates what item caused the problem.

ERROR #241: Missing operand

Generated when the Simulator does not find an operand for an operator in the USER_EXPRESSION directive. Output in the list file indicates what item caused the problem.

ERROR #242: Improper operators order

Generated when the Simulator finds that the operators in the USER_EXPRESSION directive are in the wrong order. Output in the list file indicates what item caused the problem.

ERROR #243: Two consecutive conditional expressions

Generated when the Simulator finds two consecutive conditional expressions in the USER_EXPRESSION directive. The expression defined is ignored.

ERROR #244: Undefined family name,use default family

Generated when the Simulator finds a FAMILY property name on a primitive which was not defined in the WIRE_ESTIMATE directive. The default family name is used for this primitive.

ERROR #245: Too many parameters - only 5 allowed

Generated when the Simulator finds more than five formal parameters specified in one USER_EXPRESSION directive, or more than five values in the USER_PARAMETER directive. The expression or parameter is ignored.

ERROR #246: Restore Failed, check simulator version

Generated when the Simulator detects that the RESTORE operation has failed. The SAVE file may have been created under a different version of the Simulator, the state of Realfast (ON or OFF) may be different in the SAVE file from that in the current simulation session, or the SAVE file may somehow have become corrupted.

ERROR #247: Command not supported in command script

Generated when the user attempts to invoke a command which may not be invoked from a script. Enter the command interactively.

ERROR #248: Save failed, check disk space

Generated when the Simulator detects that the SAVE command has failed. The user must have sufficient disk space in order to execute this command.

ERROR #249: Restore file not found

Generated when the Simulator is unable to find the specified RESTORE file. Check the directory name and file name.

WARNING #250: Illegal delay of zero in delay table

Generated when the Simulator finds a delay value of 0 in the delay table. Delay values must be greater than 0, and a delay of 1 ns is assumed.

ERROR #251: Delay format is not for min, typ, max.

Generated when the Simulator finds that the DELAY_MODE directive is set but that the format used for delay values is not "[min,typ,max]". Check that all parts specify delay in the proper format or eliminate the

DELAY_MODE directive.

ERRORS #252 through #255: Reserved

APPENDIX A
S-32/S-320 ADDITIONAL FEATURES

The following additional features are available on the VALID S-32/S-320 platform.

1.1 SAVE AND RESTORE FUNCTION

A Save and Restore function allows the user to Save a particular state of a simulation session, then continue with simulation, and later return to the previously saved state by using the RESTORE command or directive.

The paragraphs below describe the SAVE command, the RESTORE command, and the RESTORE directive.

SAVE COMMAND

SAVE *filename*

This command records the state of the Simulator in a file called *filename* during an interactive session. This file can later be used to RESTORE the Simulator's state. All Simulator status is output to the file, including that of Realfast if it is in use. The contents of any output files that are open at the time will not be stored in the status file, and thus cannot be RESTOREd. If *filename* already exists, it is overwritten. This command may not be invoked from a script.

This command may be used to store the status at any point during a simulation; conventional design loading can subsequently be bypassed, dramatically reducing the amount of time required to reach a previously attained simulation state. Note that a substantial amount of disk space is required in order to SAVE the Simulator's state. An even greater amount is necessary if the state includes Realfast status. The user should ensure that sufficient disk storage space exists before invoking this function.

RESTORE COMMAND

RESTORE *filename*

This command restores the SAVEd status of the Simulator from the file *filename.* After RESTOREing, the Simulator's state is the same as that when the SAVE command was originally invoked. Note that the contents of any output files which were open when the SAVE command was invoked have not been SAVEd, and the files are reset. If Realfast was in use when the SAVE command was invoked, it must be used when RESTORE is invoked. This command may not be invoked from a script.

RESTORE DIRECTIVE

This directive is used to restore the previously SAVEd status of the Simulator from the file *saved file.* The file name must be enclosed in quotes. If this directive is included, the Simulator will bypass its normal design loading and initialize to the same state as that when the SAVE command was invoked to create the specified file. Since the RESTORE operation only requires a few minutes to execute, this directive can dramatically reduce the amount of time required to initialize the Simulator to a previously attained simulation state. Here is an example:

```
RESTORE 'counter.sav';
```

If Realfast was in use when the SAVE command was invoked, it must be used when the RESTORE directive is used. Note that, with the exception of the USE_REALFAST directive, any other directive specified with the RESTORE directive will not have any effect on the simulation. To avoid confusion, it may be advisable to eliminate any other directives from the file.

1.2 SIMULATOR INTERRUPTION

The Simulator can also be interrupted during its execution. This is accomplished by pressing the backslash key (\ ) while holding the **Control** key down. The Simulator detects

this character and executes a software interrupt, stopping the command in progress. With this facility, the user is able to terminate a Simulator command before completion and regain interactive control. This feature can be used to terminate various time-consuming commands, and is particularly useful when the user detects some error. Some common applications include the following:

1. Interrupt the SIMULATE command during a long simulation interval; the screen will be updated to reflect the current status (including the current simulation time). When using Realfast, note that CTRL \ is ignored if invoked during the SIMULATE command.

2. Stop output in the echo area which is the result of a LIST command.

3. Execute a PAUSE in a script file; the script may later be RESUMEd, beginning with the command which follows the interrupted command.

4. Interrupt a command that is generating a large amount of screen data (e.g WAVEFORM, ROW, or REDISPLAY) before the entire screen is re-drawn.

Note that the CTRL \ capability is only available on the S-32 in full-screen simulation.

1.3 USER-CODED PRIMITIVES

The SCALD Logic Simulator allows users to code Simulator models in PASCAL, and refer to them using standard SCALD drawings. This section is a specification of this feature, the Simulator User-Coded Primitives or "UCPs."

The use of UCPs allows the user to expand the "parts set" understood by the Logic Simulator. A UCP has three basic parts: a body definition for drawing; a description of the "pin-out" of the part for the Simulator; and a PASCAL program to model the behavior of the part.

1.4 THE PASCAL CODE FOR SYSTEMS RUNNING UNIX

Multiple user-coded object files may be linked on the S-320 and other platforms running UNIX in the creation of a Simulator executable which contains multiple UCPs. This allows multiple users to create their own source files for UCPs, and use the object files created by others without requiring the source code.

If the user codes his primitive as a single PASCAL procedure to be linked to the Simulator, it must be of the following form:

```
unit unit_for_userprim;
interface
uses (*$U userglob.obj*) userglob;

   procedure userprim;
implementation
   procedure userprim;
   const
      { user's constant definitions, if any }
   type
      { user's type definitions, if any }
   var
      { user's var definitions, if any }
   begin
      { body of user's userprim routine }
   end;
end.
```

To compile and link "userprim.pas" with the Simulator under UNIX, type:

/u0/scald/simulator/mkucpsim userprim

This script prints any syntax errors on the screen.

The procedure userprim may use any PASCAL language features provided by SVS Pascal.

There are a number of data structure access routines provided for the user to get signal values and store signal values and to schedule simulation events. These are discussed below.

If the user has more than one UCP, separate procedures must be provided for each, nested within userprim. It is up to the user to dispatch among these several UCPs. The Simulator only calls the procedure userprim. There is an access function (get_number) provided that returns the number of the particular user primitive to be called.

1.5 THE PASCAL CODE FOR SYSTEMS RUNNING VMS

The user must code his or her primitive as a single PASCAL procedure that is linked to the Simulator. This procedure must be of the following form:

```
(*
$S-,C+,X-,W-
*)
MODULE userprim(output);

CONST
  {user constant definitions}
  MAX_PIN_BIT_NUMBER = <value of your choice>;

TYPE
  {user type definitions}

  %INCLUDE 'SYS$SCALD:USERPRIM.TYP'
  %INCLUDE 'SYS$SCALD:USERPRIM.DCL'
  [GLOBAL] PROCEDURE userprim;
        .
        .
        .
  END {of procedure userprim};

END {of module}.
```

The "(output)" parameter on the MODULE line is required if the user's PASCAL procedure includes any write or writeln statements. To compile and link "userprim.pas" with the Simulator under VMS, type:

@ SYS$SCALD:MKUCPSIM USERPRIM

This script prints any syntax errors to the screen.

The procedure userprim may use any PASCAL language features provided by the host's PASCAL dialect. However, if the user ever intends to use the Simulator under UNIX as well, the procedure should adhere to ISO-standard PASCAL.

There are a number of data structure access routines provided for the user to get signal values, to store signal values, and to schedule simulation events. These are discussed below.

If the user has more than one UCP, separate procedures must be provided for each, nested within userprim. It is up to the user to dispatch among these several UCPs. The Simulator only calls the procedure userprim. There is an access function provided that returns the number of the particular user primitive to be called.

1.6 THE PASCAL CODE FOR SYSTEMS RUNNING CMS

The user must code his primitive as a single PASCAL procedure that is linked to the Simulator. This procedure must be of the following form:

```
SEGMENT UCPSEG;

CONST
  { user constant definitions }
  MAX_PIN_BIT_NUMBER = <value of your choice>;

TYPE
  { user type definitions }

  %INCLUDE UCPTYP
  %INCLUDE UCPDCL
  PROCEDURE userprim; EXTERNAL;
  PROCEDURE userprim;
        .
        .
        .
  END { of procedure userprim };
  . { This is really a dot in the file. }
```

To compile and link "userprim pascal" with the Simulator under CMS, type:

MKUCPSIM USERPRIM

This exec will print any syntax errors to the screen.

The procedure userprim may use any PASCAL language features provided by the host's PASCAL dialect. However, if the user ever intends to use the Simulator under UNIX as well, the procedure should adhere to ISO-standard PASCAL.

There are a number of data structure access routines provided for the user to get signal values and store signal values and to schedule simulation events. These are discussed below.

If the user has more than one UCP, separate procedures must be provided for each, nested within userprim. It is up to the user to dispatch among these several UCPs. The Simulator only calls the procedure userprim. There is an access function provided that returns the number of the particular user primitive to be called.

1.7 RUNNING A SIMULATOR CONTAINING UCPs

Since a Simulator linked with UCPs is a different program than the released Simulator, it must be invoked differently. The following sections describe how to run your own Simulator under the different operating systems.

RUNNING YOUR SIMULATOR UNDER UNIX

To run your Simulator under the Graphics Editor under UNIX, start the Graphics Editor as you normally would, and EDIT your drawing. When you want to run the Simulator, type:

set user_sim *your_simulator*
simulate

where *your_simulator* is the name of your simulator. This name must be specified with its full pathname. The Simulator specified will be invoked.

To run your Simulator as a stand-alone simulator under UNIX, copy the file /usr/bin/simulate into one of your directories and edit it so that the line that begins

/u0/scald/simulator/sim

is changed to give the name of YOUR executable file. After you make this change, give the name of YOUR copy of this script when you want to run the Simulator.

RUNNING YOUR SIMULATOR UNDER VMS

Type:

@ SYS$SCALD:SIMASSIGN
RUN SIM

where SIM is the name of your version of the Simulator.

RUNNING YOUR SIMULATOR UNDER CMS

Running the SIMULATE EXEC accesses the first Simulator in your search path. If you place the disk with your Simulator earlier in your search path (e.g., on your A disk) than the disk with the release Simulator, the EXEC will use your version.

1.8 BODY DEFINITION FOR UCPs

The body definition of a UCP is nearly the same as the body definition for any other primitive part -- see Section 4 of the Library Reference Manual. There are some additional rules:

1. For every pin on the part being modeled there must be a pin on the body.

2. A vectored pin name must appear on a single pin. For example, if there is a pin name PNAME<15..0>, you must not have a pin PNAME<15..8> and another PNAME<7..0>.

3. Vectored pins must always have the most significant bit on the left.

4. A part may have up to 512 pins.

5. A pin of a part may be up to 320 bits long.

Samples of correct Simulator bodies are found in any standard Valid Simulator library (for example, 100K.SIM).

In addition to the .BODY drawing that describes the UCP, a .PRIM drawing is required to mark the UCP as a primitive for compilation. The .PRIM drawing contains a DRAWING body and a DEFINE body. The TITLE and ABBREV properties should correspond to the UCP name. The SCALD directory containing the .PRIM files must have this line

FILE_TYPE = SIM_DIR;

as the first line.

Since SCALD directories created by GED have this line

FILE_TYPE = LOGIC_DIR;

as the first line, the user must edit the SCALD directory to give it the proper FILE_TYPE. Only User-Coded Primitives should be placed in a SCALD directory with a FILE_TYPE of SIM_DIR.

1.9 UCP PINOUT DESCRIPTIONS

The Simulator must know how each of the pins of a UCP are defined. This information is specified in the user primitive configuration file. The Simulator must know:

1. The name of the UCP in the name of the .PRIM drawing.
2. The number of input and output pins.
3. The name of each pin, and if it is size-replicated.

This information is passed to the Simulator using the following format:

```
primitive 'prim name';
  pin
    INPUT_SPEC = 'string':size, 'string':size;
    INPUT_SPEC =  'string':size;
    OUTPUT_SPEC = 'string':size, 'string':size;
    OUTPUT_SPEC = 'string':size;
  end_pin;
end_primitive;
      .
      .
      .
primitive 'prim name';
  pin
    INPUT_SPEC = 'pin name':size, 'pin name':size;
    INPUT_SPEC = 'pin name':size;
    OUTPUT_SPEC = 'pin name':size, 'pin name':size;
    OUTPUT_SPEC = 'pin name':size;
  end_pin;
  OWN_STORAGE integer ;
  OWN_STORAGE_INIT integer ;
end_primitive;
END.
```

Follow these rules when making this file:

- INPUT_SPECs, OUTPUT_SPECs can be specified using a list (elements separated by commas), or with separate commands.

- All INPUT_SPECs must precede all OUTPUT_SPECs.

- *pin name* and *prim name* are strings. The single quote mark and the colon are NOT allowed in strings. A *prim name* cannot be longer than 20 characters. There is no restriction on the length of a *pin name.* The pin name should use standard SCALD Language syntax. For example, a low-asserted signal should always appear as

  - G

and not as

G *

- *size* specifies how wide the pin is. There are two forms, the first for sizeable parts, the second for parts of fixed size.

  - For sizeable parts

    size = SIZE

    and the pin has the subscript <*size* - 1 . . 0> when bit ordering is right_to_left, or the subscript <0 . . *size* - 1> when bit ordering is left_to_right.

  - For parts of fixed size

    size = *K*

    where *K* is an integer and the pin has the subscript <*K* - 1 . . 0> when bit ordering is right_to_left, or <0 . . *K* - 1> when bit ordering is left_to_right.

- If a primitive is to have "own" storage, the OWN_STORAGE command specifies the number of words and the OWN_STORAGE_INIT command gives the initialization value for the entire array.

- There is no limit to the number of UCPs a user may write.

A Simulator directive determines which user primitive configuration file is used. This is the USER_PRIM_CONFIG directive. This directive is described below.

USER_PRIM_CONFIG DIRECTIVE

This directive is used when user-coded primitives are in use. The directive specifies the name of the user primitive configuration file that contains the pin names of the user-coded primitive. The format is explained in the section on User-Coded Simulator Primitives. The filename must be quoted. Here is an example:

USER_PRIM_CONFIG '*primconf.dat*';

OWN STORAGE IN UCPs

The UCP itself is a PASCAL program (details below) that performs the simulation of the primitive. The values of local variables in the UCP will be lost from call to call. Since it is necessary to have some state preserved from call to call, the user may also specify a block of storage that is accessible only by the user primitives, and its state will be preserved from call to call. (It is analogous to an ALGOL "own" variable.) The local storage is an array of integers:

static_storage: ARRAY [1 . . *n*] OF INTEGER;

where *n* is an integer.

1.10 FUNCTIONS PROVIDED FOR USE IN UCPs

There are a variety of functions provided to facilitate coding of user-coded primitives. The following predefined types, constants, and routines are available for use in any UCPs.

Predefined constants:

MAX_PIN_BIT_NUMBER = 3199;

Predefined types:

```
LOGIC_TYPE = (LOGIC_0, LOGIC_1, LOGIC_Z,
        LOGIC_U,);
LOGIC_PIN_ARRAY = packed array [0 . . max]
```

of logic_type;

where *max* is the max_pin_bit_number.

STR20 = packed array [1..20] of char;
STR256 = packed array [1..256] of char;

Predefined routines:

FUNCTION get_number: INTEGER;

Returns the number of the primitive to be simulated on this call to userprim. The primitives are assigned successive numbers in the order in which they were defined in the user primitive configuration file, the first one being "1."

PROCEDURE get_name(VAR name: str20);

Returns the name of the primitive to be simulated on this call to userprim.

PROCEDURE get_path(VAR name: str256);

Returns the path name that uniquely determines the primitive to be simulated on this call to userprim.

FUNCTION get_size: INTEGER;

Returns the value of the size property of this primitive.

FUNCTION get_delay: INTEGER;

Returns the value of the rise or fall delay properties of this primitive, whichever is larger (equal to the delay property when rise and fall delays are not in use), in picoseconds (see put_pin).

FUNCTION get_rise: INTEGER;

Returns the value of the rise delay property of this primitive in picoseconds (see put_pin).

FUNCTION get_fall: INTEGER;

Returns the value of the fall delay property of this primitive in picoseconds (see put_pin).

PROCEDURE delay_mode(VAR rise_fall, pin_delay: boolean);

Returns the value of the simulation directive flags RISE_FALL and PIN_DELAY, respectively.

FUNCTION get_pdelay(output_index: INTEGER): INTEGER;

Returns the larger of the rise or fall pin delay properties, in picoseconds, for the current output pin which is addressed by output_index.

FUNCTION get_prise(output_index: INTEGER): INTEGER;

Returns the value of the rise delay property, in picoseconds, for the current output pin which is addressed by output_index.

FUNCTION get_pfall(output_index: integer): INTEGER;

Returns the value of the fall delay property, in picoseconds, for the current output pin which is addressed by output_index.

FUNCTION get_current_time: INTEGER;

Returns the current simulation time.

FUNCTION get_wire_delays(pin: INTEGER;
VAR rise_delay, fall_delay: REAL)
: BOOLEAN;

Returns the value of the rise and fall wire delays of the output pin, pin. Returns TRUE if pin is a legal pin number; returns FALSE if not.

The pins of a primitive are assigned successive numbers in the order in which they were defined in the user primitive configuration file. The first pin of each primitive is given the number "1." The bits of a pin are numbered from most significant to least significant as 0_ .._ LastBitNum (left-to-right ordering) or LastBitNum_ .._ 0 (right-to-left ordering).

FUNCTION get_bit_of_pin(pin, b: INTEGER;
VAR val: LOGIC_TYPE) : BOOLEAN;

Stores the value of the b bit of pin in val. Returns TRUE if pin is a legal pin number and if b is a legal bit number within that pin; returns FALSE if not.

FUNCTION get_pin(pin: INTEGER;
VAR values: LOGIC_PIN_ARRAY;):
BOOLEAN;

Stores the value of the i"th" bit of pin in the i"th" location of values. The last bit number of the pin must be less than or equal to the user-defined constant MAX_PIN_BIT_NUMBER. Returns TRUE if pin is in the range 1 . . Last pin number; returns FALSE if not.

FUNCTION put_pin(pin: INTEGER;
VAR values: LOGIC_PIN_ARRAY;
time: INTEGER): BOOLEAN;

Forces pin to assume a new value, as specified by values at the time (current simulation time + time). time is in picoseconds (pico = 10 exp -12), where 1.27 nanoseconds is 1270. Returns TRUE if pin is a legal pin number; returns FALSE if not.

FUNCTION logic_AND (a, b: LOGIC_TYPE):
LOGIC_TYPE;

Returns the "AND" of a and b.

FUNCTION logic_OR (a, b: LOGIC_TYPE):
LOGIC_TYPE;

Returns the "OR" of a and b.

FUNCTION logic_XOR (a, b: LOGIC_TYPE):
LOGIC_TYPE;

Returns the "XOR" of a and b.

FUNCTION logic_NOT (a: LOGIC_TYPE):
LOGIC_TYPE;

Returns the complement of "a."

FUNCTION logic_to_int (a: LOGIC_TYPE; VAR b: INTEGER) : BOOLEAN;

Converts "a" to an integer, returned in "b." Returns TRUE if "a" had the value logic_0 or logic_1; otherwise returns false.

FUNCTION int_to_logic (a: INTEGER; VAR b: LOGIC_TYPE) : BOOLEAN;

Converts "a" to a logic_type, returned in "b." Returns TRUE if "a" had the value 0 or 1; otherwise returns FALSE.

FUNCTION int_shift (a, n: integer) : integer;

Returns the value of the integer "a" left shifted by n bits within a host machine word. Zeros are entered into the right end. If n is negative, "a" is right shifted by -n bits and zeros are entered into the left end.

FUNCTION put_own (index, val: INTEGER) : BOOLEAN;

Puts the value, val, in the index location of the own array if it is within range. Returns TRUE if index was within range, FALSE if not.

FUNCTION get_own (index: INTEGER; VAR value: INTEGER): BOOLEAN;

Stores the contents of the index location of the own array into value. Returns TRUE if the index is within range, FALSE if not.

PROCEDURE report_error (errnum: INTEGER);

Outputs a report of the current primitive name and path with the identifying number supplied by the user in errnum.

1.11 EXAMPLE OF A USER-CODED PRIMITIVE

The following example models three parts -- an 8-function ALU, a 12-bit latch, and a 32-word memory with a clear line and separate read and write addresses.

USER CONFIGURATION FILE

The user primitive configuration file specifies for each primitive: the name of the primitive, the names and widths of the input and output pins, the amount of user storage required, and the initialization value for the user storage.

```
PRIMITIVE 'S381';

  PIN
    INPUT_SPEC  = 'a':SIZE, 'b':SIZE, 's':3, 'ci':1;
    OUTPUT_SPEC = 'f':SIZE, 'co':1, '-g':1, '-p':1;
  END_PIN;

END_PRIMITIVE;

PRIMITIVE 'LATCH12';

  PIN
    INPUT_SPEC  = 'd':12, 'enin':1, 'enout':1;
    OUTPUT_SPEC = 'q':12;
  END_PIN;

  OWN_STORAGE 1;
  OWN_STORAGE_INIT 0;

END_PRIMITIVE;

PRIMITIVE 'USERMEM';

  PIN
    INPUT_SPEC  = 'ra':5,'wa':5,'we':1,'mr':1;
    INPUT_SPEC  = 'd':SIZE;
    OUTPUT_SPEC = 'q':SIZE;
  END_PIN;

  OWN_STORAGE 2000;
  OWN_STORAGE_INIT 0;

END_PRIMITIVE;

END.
```

VMS PASCAL MODULE EXAMPLE

```
(*
*)
(*$S-,C+,X-,W-*)
MODULE USERPRIM(OUTPUT);

(***************************** CONSTANTS **********************)

CONST

      (*********************************************************
      * All user-coded primitives must define the constant    *
      * MAX_PIN_BIT_NUMBER.  This constant defines the size   *
      * of the type LOGIC_PIN_ARRAY (see [SCALD]USERPRIM.DCL) *
      * which is used whenever an array of pin values is      *
      * passed to or returned from a procedure.               *
      *********************************************************)

MAX_PIN_BIT_NUMBER = 200;


(*****************************   TYPES   **********************)

TYPE

      (*********************************************************
      *                                                       *
      * Get the user-primitive type definitions from the      *
      * SCALD library.                                        *
      *                                                       *
      *********************************************************)

%INCLUDE 'SYS$SCALD:USERPRIM.TYP'


(*********************** PROCEDURE DEFINITIONS *****************)

      (*********************************************************
      *                                                       *
      * Get the user-primitive procedure definitions from the *
      * SCALD library.                                        *
      *                                                       *
      *********************************************************)
```

```
%INCLUDE 'SYS$SCALD:USERPRIM.DCL'

      (*********************************************************
      *                                                       *
      * Do NOT include user-defined procedures here.  Make    *
      * them sub-procedures of the USERPRIM procedure.        *
      *                                                       *
      *********************************************************)


(***************************** USERPRIM ***********************)

      (*********************************************************
      *                                                       *
      * This entire module has only one procedure definition, *
      * namely USERPRIM.  All other procedures are sub-       *
      * procedures of USERPRIM.  The user is free to declare  *
      * local variables, types, constants, and procedures     *
      * within USERPRIM.  The following example of a user-    *
      * coded primitive defines a read-write 32-word memory   *
      * with a clear line, and separate read and write        *
      * addresses.  It makes use of "own-storage" to store    *
      * memory contents.                                      *
      *                                                       *
      * Throughout this example, right-to-left bit ordering   *
      * is assumed.                                           *
      *                                                       *
      *********************************************************)

[GLOBAL] PROCEDURE USERPRIM;
  CONST
      MinUserPrimNum = 1;
      MaxUserPrimNum = 3;
      BitsPerHostWord = 32;
      Debug = false;
  TYPE
      Value_Array = array [1 .. 10] of integer;
  VAR
      u_primnum,
      u_size,
      u_delay:  integer;

  procedure user_alu;

  var
    a,b,s,ci,f,co,p,g: logic_pin_array;
    select,s2,s1,s0,i: integer;
    c: logic_type;

  begin (* user_alu *)
```

```
  if not (get_pin(1,a) and    (* get inputs *)
          get_pin(2,b) and
          get_pin(3,s) and
          get_pin(4,ci) and
          logic_to_int(s[2],s2) and
          logic_to_int(s[1],s1) and
          logic_to_int(s[0],s0)) then REPORT_ERROR(1);

  select := s2*4 + s1 + s1 + s0;
  p[0]  := logic_1;
  g[0]  := logic_1;
  co[0] := logic_0;
  c := ci[0];

  for i := 0 to u_size-1 do   (* do function *)
    case select of

  0: f[i] := logic_0; (* CLEAR *)

  1: begin
     end;

  2: begin
     end;

  3: begin
       f[i] := logic_xor( logic_xor(a[i],b[i]) , c );
       c := logic_or( logic_and(a[i],b[i]) ,
                      logic_and(c , logic_or(a[i],b[i])) );
     end;

  4: f[i] := logic_xor(a[i],b[i]);    (* XOR *)

  5: f[i] := logic_or(a[i],b[i]);     (* OR *)

  6: f[i] := logic_and(a[i],b[i]);    (* AND *)

  7: f[i] := logic_not(logic_0);      (* SET *)

  end;

  co[0] := c;

  if not ( put_pin(5,f,u_delay)  and
           put_pin(6,co,u_delay) and
           put_pin(7,g,u_delay)  and
           put_pin(8,p,u_delay) ) then REPORT_ERROR(2);

end (* user_alu *);


procedure user_latch;
```

```
  const Dpin = 1; ENINpin = 2; ENOUTpin = 3; Qpin = 4;
  var  ok:    boolean;
       i,j,k: integer;
       enin,
       enout: LOGIC_TYPE;
       Dval,
       Qval:  LOGIC_PIN_ARRAY;
begin (* user_latch *)

  ok := GET_BIT_OF_PIN(ENINpin,0,enin);
  if ( not ok ) or debug then REPORT_ERROR(1);
  if enin = LOGIC_1 then
  begin
    ok := GET_PIN(Dpin,Dval);
    if ( not ok ) or debug then REPORT_ERROR(2);
    j := 0;
    for i := 11 downto 0  do
    begin
      ok := LOGIC_TO_INT(Dval[i],k);
      if ( not ok ) or debug then REPORT_ERROR(3);
      j := (j*2) + k;
    end;
    ok := PUT_OWN(1,j);
    if ( not ok ) or debug then REPORT_ERROR(4);
  end;

  ok := GET_BIT_OF_PIN(ENOUTpin,0,enout);
  if ( not ok ) or debug then REPORT_ERROR(5);
  if enout = LOGIC_1 then
  begin
    ok := GET_OWN(1,j);
    if (not ok ) or debug then REPORT_ERROR(6);
    for i := 0 to 11 do
      if odd(INTEGER_SHIFT(j,-i)) then Qval[i] := LOGIC_1
                                  else Qval[i] := LOGIC_0;
    ok := PUT_PIN(Qpin,Qval,u_delay);
    if ( not ok ) or debug then REPORT_ERROR(7);
  end;

end (* user_latch *);


procedure user_mem;
  const RApin=1; WApin=2; WEpin=3; MRpin=4; Dpin=5; Qpin=6;
  var ok:     boolean;
      v:      LOGIC_TYPE;
      v0,v1,
      adr,
      i,n:    integer;
      Dval,
      Qval,
      RAval,
```

```
    WAval:  LOGIC_PIN_ARRAY;
    temp:   Value_Array;

function Val_to_Adr(var val: LOGIC_PIN_ARRAY) : integer;
  var adr,bit,v:  integer;
      ok:         boolean;
begin (*Val_to_Adr*)
  adr := 0;
  for bit := 4 downto 0 do
  begin
    ok := LOGIC_TO_INT(val[bit],v);
    if ( not ok ) or debug then REPORT_ERROR(1);
    adr := adr*2+v;
  end;
  Val_to_Adr := adr;
end (*Val_to_Adr*);


procedure Conv_to_ValArr(var val: LOGIC_PIN_ARRAY;
                         var ValArr: Value_Array);
  var i,j,k,v: integer;
      ok:      boolean;
begin (*Conv_to_ValArr*)
  j := 1; k := 0;
  for i := 0 to u_size-1 do
  begin
    if k = 0 then ValArr[j] := 0;
    ok := LOGIC_TO_INT(val[i],v);
    if ( not ok ) or debug then REPORT_ERROR(2);
    ValArr[j] := ValArr[j]+INTEGER_SHIFT(v,k);
    if k = BitsPerHostWord-1 then
    begin j := j+1; k := 0; end
    else k := k+1;
  end;
end (*Conv_to_ValArr*);


procedure Conv_to_Val(var ValArr: Value_Array;
                      var val: LOGIC_PIN_ARRAY);
  var i,j,k,v: integer;
      ok:      boolean;
      lv:      LOGIC_TYPE;
begin (*Conv_to_Val*)
  j := 1; k := 0;
  for i := 0 to u_size-1 do
  begin
    if odd(INTEGER_SHIFT(ValArr[j],-k)) then v := 1 else v := 0;
    ok := INT_TO_LOGIC(v,lv);
    if ( not ok ) or debug then REPORT_ERROR(3);
    val[i] := lv;
    if k = BitsPerHostWord-1 then begin j := j+1; k := 0; end
    else k := k+1;
```

```
    end;
  end (*Conv_to_Val*);


begin (* user_mem *)
  n := (u_size+BitsPerHostWord-1) div BitsPerHostWord;
                (*n is the number of host words needed to store
                   one u_size-bit memory word*)

  ok := LOGIC_TO_INT(LOGIC_0,v0);
  if ( not ok ) or debug then REPORT_ERROR(4);
  v0 := -v0;      (*create host-word-long value for logic_0*)

  ok := LOGIC_TO_INT(LOGIC_1,v1);
  if ( not ok ) or debug then REPORT_ERROR(5);
  v1 := -v1;      (*create host-word-long value for logic_1*)

  ok := GET_BIT_OF_PIN(MRpin,0,v);     (*v := value of MR<0>*)
  if ( not ok ) or debug then REPORT_ERROR(6);
  if ok and (v = LOGIC_1) then       (*reset the entire memory*)
    for i := 1 to n*32 do ok := PUT_OWN(i,v0)
  else
  begin
    ok := GET_BIT_OF_PIN(WEpin,0,v); (*v := value of WE<0>*)
    if ( not ok ) or debug then REPORT_ERROR(7);
    if ok then if v = LOGIC_1 then   (*write input into memory*)
    begin
      ok := GET_PIN(WApin,WAval);
      if ( not ok ) or debug then REPORT_ERROR(8);
      if ok then ok := GET_PIN(Dpin,Dval);
      if ( not ok ) or debug then REPORT_ERROR(9);
      if ok then
      begin
        adr := Val_to_Adr(WAval);  (*convert to address of memory*)
        Conv_to_ValArr(Dval,temp); (*convert input to value array*)
        for i := 1 to n do if ok then ok := PUT_OWN(adr*n+i,temp[i]);
      end;
    end;
  end;

  ok := GET_PIN(RApin,RAval); (*read memory into output pin*)
  if ( not ok ) or debug then REPORT_ERROR(10);
  if ok then
  begin
    adr := Val_to_Adr(RAval); (*convert to address of memory*)
    for i := 1 to n do if ok then
    begin
      ok := GET_OWN(adr*n+i,temp[i]);
      if ( not ok ) or debug then REPORT_ERROR(11);
    end;
    if ok then
    begin
```

```
        Conv_to_Val(temp,Qval);
        ok := PUT_PIN(Qpin,Qval,u_delay);
        if ( not ok ) or debug then REPORT_ERROR(12);
      end;
    end;

  end (* user_mem *);

BEGIN (*USERPRIM*)
  u_primnum := GET_NUMBER; (*Get the index number of the
                              primitive called*)
  u_size := GET_SIZE;      (*Get the value of the SIZE parameter*)
  u_delay := GET_DELAY;    (*And of the DELAY parameter*)


                           (*Dispatch on u_primnum*)
  if (u_primnum >= MinUserPrimNum) and
     (u_primnum <= MaxUserPrimNum) then
  case u_primnum of
1:  user_alu;
2:  user_latch;
3:  user_mem;
  end
  else REPORT_ERROR(13);

END (*USERPRIM*);

END.
(*
*)
```

APPENDIX B
PC-AT ADDITIONAL FEATURES

The following additional features are available on all of the PC-AT platforms.

1.1 SAVE AND RESTORE FUNCTION

A Save and Restore function allows the user to Save a particular state of a simulation session, then continue with simulation, and later return to the previously saved state by using the RESTORE command or directive.

The paragraphs below describe the SAVE command, the RESTORE command, and the RESTORE directive.

SAVE COMMAND

SAVE *filename*

This command records the state of the Simulator in a file called *filename* during an interactive session. This file can later be used to RESTORE the Simulator's state. All Simulator status is output to the file. The contents of any output files that are open at the time will not be stored in the status file, and thus cannot be RESTOREd. If *filename* already exists, it is overwritten. This command may not be invoked from a script.

This command may be used to store the status at any point during a simulation; conventional design loading can subsequently be bypassed, dramatically reducing the amount of time required to reach a previously attained simulation state. Note that a substantial amount of disk space is required in order to SAVE the Simulator's state. The user should ensure that sufficient disk storage space exists before invoking this function.

RESTORE COMMAND

RESTORE *filename*

This command restores the SAVEd status of the Simulator from the file *filename.* After RESTOREing, the Simulator's state is the same as that when the SAVE command was originally invoked. Note that the contents of any output files which were open when the SAVE command was invoked have not been SAVEd, and the files are reset. This command may not be invoked from a script.

RESTORE DIRECTIVE

This directive is used to restore the previously SAVEd status of the Simulator from the file *saved file.* The file name must be enclosed in quotes. If this directive is included, the Simulator will bypass its normal design loading and initialize to the same state as that when the SAVE command was invoked to create the specified file. Since the RESTORE operation only requires a few minutes to execute, this directive can dramatically reduce the amount of time required to initialize the Simulator to a previously attained simulation state. Here is an example:

```
RESTORE 'counter.sav';
```

Any other directive specified with the RESTORE directive will not have any effect on the simulation. To avoid confusion, it may be advisable to eliminate any other directives from the file.

INDEX

1 of 8 decoder primitive, 9-23
8-bit decoder primitive, 9-24
8-bit prio encoder primitives, 9-23

ABBREV property, 9-2, A-9
adder primitives, 9-18
alu primitives, 9-18
and primitive, 9-3
arithmetic primitives, 9-18
assertions, clock, 1-19
ASSERTIONS command, 7-1

batch simulation, 5-1
bidirectional nets, 1-4
BINARY_TRACE directive, 2-1
breakpoints, 1-11, 4-1
BUBBLE command, 9-1
buffer primitive, 9-5
BUS command, 7-2
Bus mode, 1-2, 1-6, 1-12, 3-1
    signal names in, 1-14
bus signals, 1-15

carry save adder primitives, 9-20
CHANGE command, 9-21, 9-22
character graphics, 1-16
circuit initialization, 1-2
CLEAR BREAKPOINT command, 4-4, 7-2
CLEAR ENABLE command, 4-3, 7-2
CLEAR PATCH command, 4-7, 7-2
clock assertions, 1-19
CLOCK command, 7-3
clock intervals, 1-10
clock period, 1-10, 2-1, 2-2
CLOCK_INTERVALS directive, 2-1
CLOCK_ON_DRIVEN directive, 2-2
CLOCK_PERIOD directive, 2-2
cmpexp.dat file, 1-8, 1-9, 2-3, 2-9

cmplst.dat file, 1-9
cmpsyn.dat file, 1-8, 1-9, 2-10, 2-13
CMS, 1-20, 1-22, A-6
command file, 1-1, 1-20, 1-21
command line, 1-8
commands
    ASSERTIONS, 7-1
    BUS, 7-2
    CLEAR BREAKPOINT, 4-4, 7-2
    CLEAR ENABLE, 4-3, 7-2
    CLEAR PATCH, 4-7, 7-2
    CLOCK, 7-3
    COMPARE, 7-3
    COVERAGE, 7-3, 7-7, 7-22
    CURSOR, 1-11, 2-7, 3-1, 3-3, 7-3
    DELTA_TIME, 3-1, 3-4, 7-4
    DEPOSIT, 1-19, 1-2
    DISPLAY, 7-5
    DUMPMEMORY, 6-1, 7-6
    EQUATE ENABLE, 4-3, 4-5, 7-6
    EQUATE PATCH, 4-8, 7-6
    EXIT, 1-23, 7-6
    HARDCOPY, 7-6, 7-16
    HISTORY, 1-2, 2-7, 3-1, 3-3, 7-7
    INIT_COVERAGE, 7-7
    INTERVAL, 7-7
    LATCH ENABLE, 4-3, 7-7
    LATCH PATCH, 4-8, 7-8
    LIST BREAKPOINTS, 4-4, 7-8
    LIST DEPOSITS, 7-8
    LIST ENABLES, 4-4, 7-8
    LIST PATCHES, 4-8, 7-8
    LIST SIGNALS, 7-8
    LIST TRACES, 7-8
    LOADMEMORY, 7-9
    LOGIC_INIT, 1-18, 4-7, 7-9
    MEMLOAD, 6-1, 6-2, 7-9
    MEMPATH, 1-11,1-18,6-1, 7-10
    MEM_INIT, 1-18, 7-9
    MOVE, 7-10
    NEXTMEMORY, 7-10
    OPEN, 1-2, 1-22, 3-1, 3-4, 7-10
    OPENMEMORY ADDRESS, 7-11

OPENMEMORY, 1-18, 1-19
PAUSE, 7-3, 7-12, 7-15
PEEK, 2-10, 7-12
PERIOD, 2-2, 7-12
PLOT, 1-13, 1-20, 1-21, 7-12
RADIX, 7-13
RECORD_ALL, 7-13
RECORD_SIGNALS, 7-13
REDISP, 7-13
REMOVE, 3-5, 7-14
RESTORE, A-2, B-2
RESUME, 7-3, 7-14, 7-15
ROW, 3-1, 3-5, 7-14
SAMPLE ENABLE, 4-3, 7-14
SAMPLE PATCH, 4-7, 7-14
SAVE, A-1, A-2, B-1, B-2
SCOPE, 1-11, 7-15
SCRIPT, 1-21, 2-3, 2-6, 7-15
SCROLL, 3-1, 3-5, 7-15
SET, 7-16
SET BREAKPOINT, 4-3, 4-4, 7-15
SET ENABLE, 4-2, 7-16
SET PATCH, 4-7, 7-16
SHOW, 7-17
SIMULATE, 1-2, 1-19, 2-7, 7-17
SNAPSHOT, 1-20, 7-18
SPACING, 3-1, 3-5, 7-18
STEP, 7-18
SYSTEM, 7-18
TERMINAL, 7-18
TRACE, 7-19
TRACE_ALL, 7-20
TRACE_CLOSE, 7-20
TRACE_INTERVAL, 5-7, 7-20
TRACE_MEM, 7-20
TRACE_OPEN, 7-20
TRACE_RADIX, 7-20
TRACE_READ, 5-7, 7-21
TRACE_RESET, 5-7, 5-8, 7-21
TRACE_START, 7-21
TRACE_STOP, 7-21
UNDO_DEPOSIT, 7-21
UPDATE_INTERVAL, 7-22

WAVEFORM, 1-16, 2-7, 3-1, 3-2, 7-2, 7-22
WIRE_DELAYS, 7-22, 8-15, 8-17
WRITE_COVERAGE, 7-3, 7-22
COMMAND_FILE directive, 1-1, 1-21, 2-3, 2-6
comments in files, 1-8
comparator primitives, 9-20
COMPARE command, 7-3
COMPERR program, 1-9
Compiler
invoking, 1-7
output files, 2-3, 2-10, 2-13
Compiler directives file, **see** files, Compiler directives
Compiler expansion file, **see** files, Compiler expansion
Compiler synonym file, **see** files, Compiler synonym
COMPILER_OUTPUT directive, 1-8, 1-9, 2-3, 2-10
component delay, 8-7
counter primitive, 9-16
counter/shift register primitive, 9-16
COVERAGE command, 7-3, 7-7, 7-22
cursor, 1-12
CURSOR command, 1-11, 2-7, 3-1, 3-3, 7-3

DECAY_TIME directive, 1-3, 2-3, 2-7
DEFAULT_DRIVE directive, 2-3, 8-10, 8-11, 8-12
DEFINE body, 9-2, A-9
delay equations user-defined, 2-13, 2-14
delay estimator, 2-3, 2-14, 8-1, 8-6, 8-7, 8-8, 8-9, 8-10, 8-12
delay properties, 8-1
DELAY property, 8-1, 8-7 9-2
delays
component, 8-1, 8-7
minimum, typical, maximum, 2-4, 8-1, 8-15
pin-to-pin, 2-6, 8-1, 8-2
ranges in,8-15
rise/fall, 2-8
DELAY_EQ property, 8-12, 8-13
DELAY_ESTIMATOR directive, 2-4, 8-7, 8-10
DELAY_MODE directive, 2-4, 8-1
DELAY_PARAM property, 8-12, 8-13, 8-14
DELTA_TIME command, 3-1, 3-4, 7-4
DEPOSIT command, 1-2, 1-19
DEPTH property, 9-14

directives
BINARY_TRACE, 2-1
CLOCK_INTERVALS, 2-1
CLOCK_ON_DRIVEN, 2-2
CLOCK_PERIOD, 2-2
COMMAND_FILE, 1-1, 1-21, 2-3, 2-6
COMPILER_OUTPUT, 1-8, 1-9, 2-3, 2-10
DECAY_TIME, 1-3, 2-3, 2-7
DEFAULT_DRIVE, 2-3, 8-10, 8-11, 8-12
delay estimator, 8-7
DELAY_ESTIMATOR, 2-4, 8-7, 8-10
DELAY_MODE, 2-4, 8-1
EXP_EVALUATOR, 2-5, 8-12
MEM_STATE, 2-5, 9-14
OUTPUT, 1-20, 2-5, 2-10
PIN_DELAY, 2-6, 8-3 8-4 8-5
REAL_CHIP_LIBRARY, 2-6
REMOTE_HOST, 2-7
RESOLUTION, 1-10, 1-11, 2-7, 3-1, 3-3
RESTORE, A-2, B-2
RISE_FALL, 2-8, 8-2, 8-3, 8-5
ROOT_DRAWING, 1-7, 1-8, 2-3, 2-9, 2-10
SESSION_LOG, 1-20, 2-10
SIGNAME_CHARS, 2-10
SYNONYM_FILE, 1-8, 1-9, 2-3, 2-10
TABULAR_TRACE, 2-11, 5-1, 5-8
TERMINAL, 2-11
TIMING_CHECK, 2-12, 9-20
TRACE_RADIX, 2-12
USER_EXPRESSION, 2-13, 8-12, 8-13
USER_PARAMETER, 2-14, 8-12, 8-13, 8-14
USER_PRIM_CONFIG, A-12, A-13
USE_IF, 2-12
USE_REALFAST, 2-13
USE_SYNONYM, 2-13
WIRE_DELAYS, 2-14, 8-6 8-15
WIRE_ESTIMATE, 2-14, 8-9, 8-11
families in, 8-9
directives files, 2-1
Compiler, 1-7
Simulator, 1-7
directories SCALD, A-9
DISPLAY command, 7-5

display screen
    echo area,1-9, 1-11
    status line,1-9, 1-10
DRAWING body, 9-2, A-9
drawing name, 9-1
drawing type .PRIM, A-9
drive, 2-3
DRIVE property, 8-8, 8-10, 8-12, 8-13, 8-14
drivers, unidirectional, 1-4
DUMPMEMORY command, 6-1, 7-6

edge to edge primitives, 9-22
encoder and decoder primitives, 9-23
EQUATE ENABLE command, 4-3, 4-5, 7-6
EQUATE PATCH command, 4-8, 7-6
error messages, 1-11, 10-1
errors
    Compiler, 1-7, 1-9
    report of, 1-21
    simulation, 10-1
EXIT command, 1-23, 7-6
expression evaluator, 2-5, 2-13, 2-14, 8-1 8-12
EXP_EVALUATOR directive, 2-5, 8-12

FALL property, 8-1, 8-7
FAMILY property, 2-15, 8-9
file names, 1-22
files
    command, 1-1, 1-20, 1-21
    Compiler directives, 1-7
    Compiler expansion, 1-8, 1-9, 2-3, 2-9
    Compiler listing, 1-9
    Compiler output, 2-3, 2-10, 2-13
    Compiler synonym, 1-8, 1-9, 2-10, 2-13
    Directives, 2-1
    input command, 2-6
    input, 1-22
    listing, 1-20, 2-5, 2-10
    log, 1-20, 1-21
    output command, 2-5
    output, 1-20, 1-22
    Simulator directives, 1-7
    user primitive configuration, A-10, A-19

waveform input, 1-20, 1-21
wire delay, 2-14, 8-7, 8-15

gates, phantom, 8-9, 8-11
GED, 9-1
Graphics Editor, 9-1

HARDCOPY command, 7-6, 7-16
HIGH property, 9-22
HISTORY command, 1-2, 2-7, 3-1, 3-3, 7-7
HOLD property, 9-21

identity primitive, 9-6
initialization
of circuits, 1-2
of signals, 1-18
INIT_COVERAGE command, 7-7
interrupt feature, A-2
INTERVAL command, 7-7

jk primitive,9-6

LATCH ENABLE command, 4-3, 7-7
LATCH PATCH command, 4-8, 7-8
latch primitive, 9-7, 9-8, 9-9
librarian, 9-1
libraries SIM, 9-1
library development, 8-1, 9-1
LIST BREAKPOINTS command, 4-4, 7-8
LIST DEPOSITS command, 7-8
LIST ENABLES command, 4-4, 7-8
LIST PATCHES command, 4-8, 7-8
LIST SIGNALS command, 7-8
LIST TRACES command, 7-8
listing file, 1-20, 2-10
load calculation, 8-9
loading, 8-6, 8-7, 8-8, 8-9, 8-10
LOADMEMORY command, 7-9
LOAD_FACTOR property, 8-9, 8-11, 8-12, 8-14
log file, 1-20, 1-21
logic patching, 4-1, 4-7
LOGIC_INIT command, 1-18, 4-7, 7-9
lookahead primitives, 9-19

LOW property, 9-22

MAX property, 9-22
MEMLOAD command, 6-1, 6-2, 7-9
memories, 1-18, 2-5, 5-7, 9-14
    four-state, 2-5, 9-14
    loading, 6-1
    two-state, 2-5, 9-14
memory primitives, 5-4, 6-2, 9-14
memory strength signals, 1-3, 1-4
MEMPATH command, 1-11, 1-18, 6-1, 7-10
MEM_INIT command, 1-18, 7-9
MEM_STATE directive, 2-5, 9-14
MIN property, 9-22
min pulse width primitives, 9-22
MOVE command, 7-10
mux primitive, 9-13

nets bidirectional, 1-4
NEXTMEMORY command, 7-10

open collector, 9-25
OPEN command, 1-2, 1-22, 3-1, 3-4, 7-10
open emitter, 9-25
OPENMEMORY ADDRESS command, 7-11
OPENMEMORY command, 1-18, 1-19
operating systems CMS, 1-20, 1-22, A-6
    UNIX, 1-20, 1-22, A-4
    VMS, 1-20, 1-22, A-5
or primitive, 9-4
output
    open collector, 9-25
    emitter, 9-25
    pole, 9-26
    tri-state , 9-25
OUTPUT directive, 1-20, 2-5, 2-10
output files, 1-20, 1-21, 2-3, 7-15
OUTPUT_TYPE property, 9-25

parity primitive, 9-24
parts zero-delay, 1-19
PASCAL, 9-25, A-3, A-13
pass transistor primitive, 1-4, 9-25

PAUSE command, 7-3, 7-12, 7-15
PDELAY property, 8-1, 8-2
PEEK command, 2-10, 7-12
PERIOD command, 2-2, 7-12
permissions, 9-1
PFALL property, 8-1, 8-2
phantom gates, 8-9, 8-11
physical design system, 8-6, 8-15
PIN_DELAY directive, 2-6, 8-3, 8-4, 8-5
PLOT command, 1-13, 1-20, 1-21, 7-12
plotsig.dat file, 1-20, 1-21
Plottime, 1-13, 1-21
PRIM drawing, A-9
primitives
    1 of 8 decoder, 9-23
    8-bit decoder, 9-24
    8-bit prio encoder, 9-23
    adder, 9-18
    alu, 9-18
    and, 9-3
    arithmetic, 9-18
    buffer, 9-5
    carry save adder, 9-20
    comparator, 9-20
    counter, 9-16
    counter/shift register, 9-16
    edge to edge, 9-22
    encoder and decoder, 9-23
    identity, 9-6
    jk, 9-6
    latch, 9-7, 9-8, 9-9
    lookahead, 9-19
    memory, 5-4, 6-2, 9-14
    min pulse width, 9-22
    mux, 9-13
    or, 9-4
    parity, 9-24
    pass transistor, 1-4, 9-25
    priority encoder, 9-23
    reg, 9-9, 9-10, 9-11, 9-12, 9-13
    resistor, 1-4, 9-24
    setup hold, 9-20, 9-21
    shift register, 9-16

simulation, 8-9, 9-1
timing checker, 2-12, 9-20, 9-22
ts buf, 9-5
uni pass transistor, 9-25
user-coded, 9-25, A-3
xor, 9-4
priority encoder primitive, 9-23
PRISE property, 8-1, 8-2
procedure, userprim, A-5, A-6, A-7
properties
ABBREV, 9-2, A-9
DELAY, 2-4, 2-8, 8-1, 8-7, 9-2
DELAY_EQ, 8-12, 8-13
DELAY_PARAM, 8-12, 8-13, 8-14
DEPTH, 9-14
DRIVE, 2-4, 8-8, 8-10, 8-12, 8-13, 8-14
FALL, 2-4, 2-8, 8-1, 8-7
FAMILY, 2-15, 8-9
HIGH, 9-22
HOLD, 9-21
LOAD_FACTOR, 8-8, 8-9, 8-11, 8-12, 8-14
LOW, 9-22
MAX, 9-22
MIN, 9-22
OUTPUT_TYPE, 9-25
PDELAY, 2-8, 8-1, 8-2
PFALL, 2-8, 8-1, 8-2
pin delay, 2-6
PRISE, 2-8, 8-1, 8-2
RISE, 2-4, 2-8, 8-1, 8-7
SETUP, 9-21
SIZE, 1-1, 8-11, 9-2, 9-14, A-12, A-14
TIMES, 1-1, 8-8, 8-11, 8-12
TITLE, 9-2, A-9

radix, 1-10, 1-13, 7-13
strength, 1-13
RADIX command, 7-13
Realchip, networked, 2-7
Realfast, 1-19, 2-13
Realmodel, 2-13
networked, 2-7
REAL_CHIP_LIBRARY directive, 2-6

RECORD_ALL command, 7-13
RECORD_SIGNALS command, 7-13
REDISP command, 7-13
register primitive, 9-9, 9-10, 9-11, 9-12, 9-13
REMOTE_HOST directive, 2-7
REMOVE command, 3-5, 7-14
resistor primitive, 1-4, 9-24
resolution, 1-10
RESOLUTION directive, 1-10, 1-11, 2-7, 3-1, 3-3
RESTORE command, A-2, B-2
RESTORE directive, A-2, B-2
RESUME command, 7-3, 7-14, 7-15
RISE property, 8-1, 8-7
RISE_FALL directive, 2-8, 8-2, 8-3, 8-5
ROOT_DRAWING directive, 1-7, 1-8, 2-3, 2-9, 2-10
ROW command, 3-1, 3-5, 7-14

SAMPLE ENABLE command, 4-3, 7-14
SAMPLE PATCH command, 4-7, 7-14
SAVE command, A-1, A-2, B-1, B-2
save/restore, A-2, B-2
    with Realfast, A-2
scalar signals, 1-14
SCALD Directories, 1-7, A-9
SCALD Language, 1-1, 1-17
scale, see resolution, 1-10
scale
    of simulation display, 2-7
SCOPE command, 1-11, 7-15
SCRIPT command, 1-21, 2-3, 2-6, 7-15
SCROLL command, 3-1, 3-5, 7-15
sentinels, in value file, 5-2, 5-5
SESSION_LOG directive, 1-20, 2-10
SET command, 7-16
SET BREAKPOINT command, 4-3, 4-4, 7-15
SET ENABLE command, 4-2, 7-16
SET PATCH command, 4-7, 7-16
setup hold primitives, 9-20, 9-21
SETUP property, 9-21
shift register primitive, 9-16
SHOW command, 7-17
signal assertions, 2-2
signal history, 1-2, 1-9, 1-13, 1-16, 3-5, 7-2

signal initialization, 1-18
signal mapping, 5-2
signal mapping file, 5-2, 5-3
signal name syntax, 1-1, 1-17, 7-1, A-11
signal names on Simulator display, 2-10
signal states, 1-2, 1-4, 1-5
signal strength, 1-2, 1-3, 1-4, 1-13, 9-24
    memory, 2-3
signal synonyms, 1-18
signal transitions, 1-15, 1-17
signal values, 1-2, 1-3, 1-4, 1-5, 1-9, 1-19
    high impedance/unknown, 1-3
signals
    bus, 1-15
    driven, 2-2
    low-asserted, 1-18, 9-3, A-11
    MOS, 2-3
    scalar, 1-14
    undriven, 2-2
    unnamed, 1-22
    waveforms, 1-21
SIGNAME_CHARS directive, 2-10
SIM library, 9-1
simcmd.dat file, 1-1, 1-20, 1-21, 2-3, 2-5, 7-15
simlog.dat file, 1-20, 1-21
simlst.dat file, 1-20, 2-5, 2-10
SIMULATE command, 1-2, 1-19, 2-7, 7-17
    in GED, 1-22
simulate.cmd file, 2-1
simulation
    batch mode, 1-1, 1-21, 2-3, 2-12, 5-1
    Bus mode, 1-2, 1-6
    full-screen, 1-6
    interactive, 2-12
    mainframe host, 1-6
    split-screen, 1-6, 1-22
    Waveforms mode, 1-2, 1-6, 1-9, 1-12, 2-7, 2-10, 3-1, 3-2
    with UCPs, 1-6, A-8
simulation models, 8-1, 8-7, 9-1
simulation primitives, 8-9, 9-1
    bubbled pins on, 9-1
simulation time, 1-10
Simulator, invoking, A-8

SIZE property, 1-1, 8-11, 9-2, 9-14, A-12, A-14
SNAPSHOT command, 1-20, 7-18
softkeys, 1-23
SPACING command, 3-1, 3-5, 7-18
split-screen simulation, 1-22
step, simulate for, 1-19, 7-18
STEP command, 7-18
SYNONYM_FILE directive, 1-8, 1-9, 2-3, 2-10
syntax, 4-6
SYSTEM command, 7-18
systems, operating **see** operating systems

tabular I/O, 5-6, 5-7
tabular trace format, 5-1
TABULAR_TRACE directive, 2-11, 5-1, 5-8
TERMINAL command, 7-18
TERMINAL directive, 2-11
terminal types, 1-12
TIMES property, 1-1, 8-8, 8-11, 8-12
timing assertions, 2-2, 7-1
timing checker primitives, 9-20
timing errors, 2-12
TIMING_CHECK directive, 2-12, 9-20
TITLE property, 9-2, A-9
totem pole output, 9-26
trace, tabular, 5-6
TRACE command, 7-19
trace format
    standard, 5-1
    tabular, 5-1
TRACE_ALL command, 7-20
TRACE_CLOSE command, 7-20
TRACE_INTERVAL command, 5-7, 7-20
TRACE_MEM command, 7-20
TRACE_OPEN command, 7-20
TRACE_RADIX command, 7-20
TRACE_RADIX directive, 2-12
TRACE_READ command, 5-7, 7-21
TRACE_RESET command, 5-7, 5-8, 7-21
TRACE_START command, 7-21
TRACE_STOP command, 7-21
tracing, 5-1
    value information for, 5-2

tri-state output, 9-25
trigger, 1-11, 1-12
ts buf primitive, 9-5

UCP (user coded primitive), 1-6, 9-25, A-3, A-13
ucp functions, A-5, A-6, A-7, A-13, A-14, A-15, A-16, A-17, A-
UNDO_DEPOSIT command, 7-21
uni pass transistor primitive, 9-25
UNIX, 1-20, 1-22, A-4
unnamed signals, 1-22
UPDATE_INTERVAL command, 7-22
user primitive configuration file, A-10, A-19
user-coded primitives, 9-25, A-3, A-13
userprim procedure, A-5, A-6, A-7
USER_EXPRESSION directive, 2-13, 8-12, 8-13
USER_PARAMETER directive, 2-14, 8-12, 8-13, 8-14
USER_PRIM_CONFIG directive, A-12, A-13
USE_IF directive, 2-12
USE_REALFAST directive, 2-13
USE_SYNONYM directive, 2-13

value file, 5-2, 5-5
    for tracing, 5-2
    sentinels in, 5-2, 5-5
vectors, 1-15
VMS, 1-20, 1-22, A-5

waveform input file, 1-20, 1-21
waveforms, 1-21
WAVEFORMS command, 1-16, 2-7, 3-1, 3-2, 7-2, 7-22
Waveforms mode, 1-2, 1-6, 1-9, 1-12, 2-7, 2-10, 3-1, 3-2
window size, 1-10
windows, 1-7
wire delay file, 2-14, 8-7, 8-15
wire delays
    estimated, 8-6, 8-7, 8-8, 8-9, 8-10
    feedback of, 8-6, 8-15
    load dependent, 8-6, 8-7, 8-8, 8-10
wire gates, 8-8, 8-11
wire stops, 8-6, 8-7, 8-9, 8-10
WIRE_DELAYS command, 7-22, 8-17, 8-15
WIRE_DELAYS directive, 2-14, 8-6, 8-15
WIRE_ESTIMATE directive, 2-14, 8-9, 8-11

WRITE_COVERAGE command, 7-3, 7-7, 7-22

xor primitive, 9-4
zero-delay loops, 1-19