CHAPTER 7

LOGIC SIMULATOR

7.1 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.

The graphic input language (SCALD notation) lets the designer refer to multibit-wide components and buses as single entities. The Compiler expands the design using components that are drawn from a component library until the design is expressed entirely in terms of simulation primitives. Previously, the Compiler was explicitly invoked to generate design files which were then read by the Logic Simulator; with the advent of ValidSIM, the Simulator is now able to invoke the Compiler directly or use the traditional Compiler design files.

The Timing Verifier ensures that the circuit is free of race conditions, setup and hold-time errors, pulse width errors, and clock glitches.

The Logic Simulator initializes the system to a fixed state and waits for a command from the user. The user may enter commands directly or through a command file 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 by the user through a command file, data file, or in many cases, by simulating an additional circuit specifically drawn to provide the stimulus.

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 verified to ensure that it is still working correctly after design modifications.

Although designed primarily as an interactive tool, the Logic Simulator may be run as a batch process. All commands of a simulation session may be entered into a single command file and the name of this command file included in the Logic Simulator directives file (a special file that directs the simulation process). The directives are described later in this chapter.

7.2 GETTING STARTED

The Simulator can be run on any of Valid's three supported hardware configurations: Digital VAX, IBM 370 (or equivalent), or Valid's own S-32 computing system (or equivalent - Valid's SS IV is generally considered equivalent). The program can be run from a terminal connected to these three systems or directly from the SCALD Design Station.

The design station can access the Simulator in three different ways. When connected to a host computer, the Simulator can be run from the design station using the terminal emulation mode (see Display Manager in Chapter 2). If simulation is desired under UNIX on Valid's S-32 computer, the design station can be used to run the Simulator in a full or partial screen window, either with or without using graphics capabilities. Or, the program can be invoked while running the Graphics Editor (see Graphics Editor in Chapter 3). To run the Simulator from any of the above configurations, the user simply types:

simulate

If the Simulator is invoked in a partial screen UNIX window, the window must be at least a minimum size. Under GED, this size is 48 x 86 characters. The Simulator without graphics can be run in any window at least 12 x 80; with graphics, the minimum window size is 14 x 86. The user is prevented from running the Simulator in any window which is smaller than required.

The Simulator starts by reading the Simulator directives file. The Simulator looks for this file under a specific name. For the three hardware configurations, the name must appear as follows:

simulate.cmd - For VAX and S-32 configurations
simulate cmd - For IBM or equivalent configurations

If the ROOT_DRAWING directive is specified in this file or on the simulate command line, the Simulator will invoke the Compiler directly to process the design and interpret the data during its initialization phase. If this directive is not used, the Simulator requires two files created by the Compiler during some previous invocation - the expansion file and the synonyms file; if the Simulator does not find the names of these files in the directives file, the default names "cmpexp.dat" and "cmpsyn.dat" are assumed.

7.3 DISPLAY FORMATS

The Simulator has two output formats. These are BUS mode and WAVEFORMS mode. BUS mode simultaneously displays the current value of a large group of signals selected by the user. WAVEFORMS mode can manipulate up to 200 signals; the number of signals 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
Cluster/GCluster	48
Cluster running GED	12
Ann Arbor	34
VT-100	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 can be displayed as a waveform such as produced by a standard logic analyzer or can be written to a file for subsequent input to the Plottime program. (The Plottime program uses the signal history to produce timing diagrams for display by the Graphics Editor.)

The display screen is divided into three parts: the echo area, the status lines, and the main display. The echo area shows what has been typed recently by the user. The status lines are at fixed locations on the top of the screen and are updated periodically by the Simulator. The main display shows the values of signals selected by the user and, in WAVEFORMS mode, their history.

On a cluster terminal (not under GED), the size of the main display area varies with the size of the window in which the Simulator is invoked. Not only will the number of lines increase (for up to 48 waveforms in a full-screen window), but the width of the display area will also grow 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 BUS mode names and values.

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. 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 <carriage return> before continuing to allow users to interpret the output before it is scrolled away; note that all other input is ignored until a <carriage return> is entered. On other terminal types, scrolling can be inhibited through the use of the appropriate terminal key (e.g., ctrl-S).

STATUS LINES

```
Time: xxxxxx  Step: xxx  Radix: xx  Clock: xxx / xxx  Top Row: xxx
Mem path: xxxxxxxxxx                                  Scale:  xxx.x
Scope: xxxxxxxxxx
```

These lines are a permanent part of the Simulator display. "Mem path" and "Scope" are displayed only in BUS mode, and "Top Row" is displayed only in WAVEFORMS mode. "Scale" is displayed near the top of the screen in BUS mode and near the bottom in WAVEFORMS mode.

"Time" is the current simulation time in nanoseconds. "Step" is some interval in nanoseconds which is stepped off when the SIMulate Step command is input. "Radix" shows the current radix value, which can be binary, octal, decimal, hexadecimal, or strength. "Clock" indicates the clock cycle time in nanoseconds and the number of periods into which the clock has been subdivided. "Top Row" indicates the row number of the top row on the display in WAVEFORMS mode. "Mem path" is a memory pathname that indicates the result of the last successful MEMPATH command (see below). "Scale" is the scale factor associated with the RESOLUTION command. "Scope" shows the default path name that may be set with the SCOPE command.

VALUES

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. 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.h0.s0.sl.m0..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.

OPENING SIGNALS

When the user "opens" a signal name, that signal and its value appear in the main display. A signal name is in standard SCALD syntax, except that bit lists and step values are not permitted. Names are right-justified in WAVEFORMS mode. The value of the signal appears left-justified in the current radix. The currently open signal is indicated by a "->" preceding the signal value. 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, ADR BUS<0..31> may also be known as SYSTEM BUS<40..71> and also as CHIP SELECT<0..5>:MEMORY ADR<0..27>. 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 Commands section) and then the OPENMEMORY command is used to open the desired addresses. This sequence causes an entry, which consists of the memory pathname followed by the address (enclosed in parentheses) and the value stored in the memory at that address (right justified in the current radix), to be made in the main display. Bit ranges are not currently permitted in memory displays.

BUS MODE

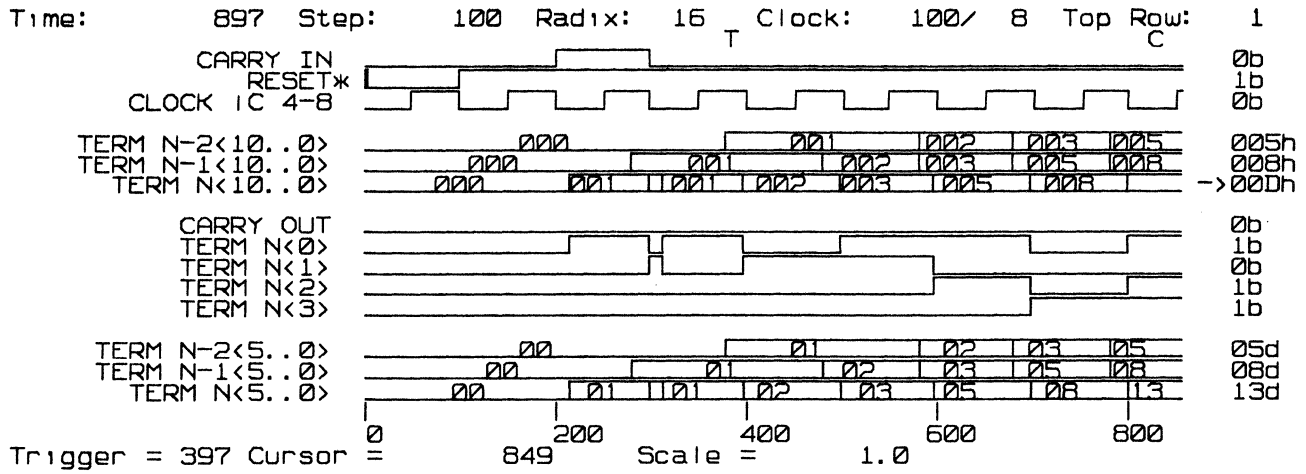
In BUS mode, the main display area is a table of signals and their values. Since this mode can display many signals on each line, a signal might not fit if its name and value require too much room horizontally. If this happens, it is still possible to open the signal as several subranges.

WAVEFORMS MODE

WAVEFORMS mode displays the history of selected signals as waveforms, with the value of buses displayed wherever possible. This mode takes advantage of the graphics capabilities of the design station by building waveforms from line segments. This mode may also be run on an ASCII or EBCDIC terminal, in which case waveforms are built from characters, or on the SS IV, in which case waveforms are built using a graphical character set.

Simulator Screen Format

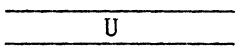
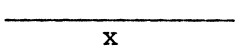
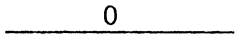
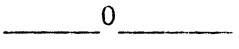
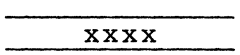
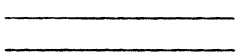
A sample screen format is shown:



The "T" character marks the time of the last encountered (Triggered) breakpoint. The "C" character marks the time at which the cursor is placed. If the character column for the "T" and "C" indicators coincide, a "B" (for both) will be output instead. 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.

Waveform Representations

When the cluster's graphics capabilities are utilized (as is the case when the Simulator is invoked with the GCLUSTER terminal type or when run under GED), waveform representations differ from when terminal characters are used. The waveform representation on the SS IV differs still again; it is almost a combination of the following two, in which characters are used, but they are from a graphical character set.

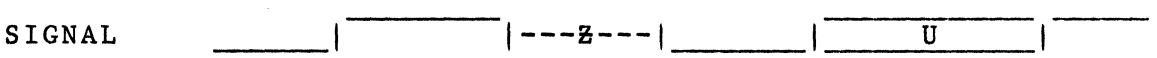
	GRAPHICS	CHARACTERS
Single bit (scalar) signals are represented as follows:		
SIGNAL = 1		^^^^^^^^^^^^
SIGNAL = 0		
SIGNAL = Z		-----Z------
SIGNAL = U		=====U======
Multiple bit buses are represented as follows:		
BUS with all bits 1		^^^^^x^^^^^^
BUS with all bits 0		
BUS with all bits Z		-----Z------
BUS with value "xxxx"		====xxxx====
BUS with value too large to fit in display space		==========

When utilizing 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. With character output, 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)

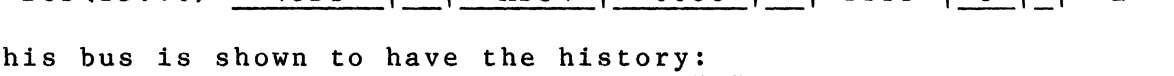
A multiple transition symbol is displayed only if some bit of the signal has changed two or more times during the time mapping to the display position. If a bus undergoes several transitions, but each bit changes only once, a multiple transition has not occurred.

A typical scalar signal might be displayed as follows:



This signal is shown to have the history: 0,1,Z,0,U,1.

A typical bus (vector) signal might be displayed as follows:



This bus is shown to have the history:
43D1,?,AD34,0000,?,FFFF,U,?,Z (where "?" indicates that the value is too large to fit in the display space).

7.4 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 the 12 states that are directly viewable by the user. The eight states that are not directly viewable are special forms of the UNDEFINED value which are sometimes used internally.

Each state may be divided into two parts, a VALUE and a STRENGTH. The VALUE of a signal is its logical level. The possible VALUEs 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	Was driven to level, now holding due to charge storage
INDETERMINATE	Could be HARD, SOFT or MEMORY

MEMORY STRENGTH signals maintain their VALUE for a limited period of time and then assume UNDEFINED VALUE. This 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 which is set with the directive:

DECAY_TIME <value in nanoseconds>

The combination of each VALUE with each STRENGTH gives the 12 viewable states. 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 A or B 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 table lists all combinations of two states.

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

7.5 INITIALIZATION OF SIGNALS AND MEMORIES

At the start of simulation, each signal is set to the undefined state. The LOGIC_INIT and MEM_INIT commands may be 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. If a memory has a bubbled output, then it has an asserted state of zero, otherwise it is one.

7.6 CHANGING SIGNAL VALUES

The user may change the value of any signal, whether driven or undriven. If the user changes the value of a signal that is driven, the signal value specified temporarily overrides the value given in the circuit that drives that signal. Whenever a signal name has a clock assertion (!P or !C) and is not driven by an output in the circuit, the signal generated by the Logic Simulator will have the timing behavior corresponding to the clock notation following the !P or !C.

7.7 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.

7.8 SIMULATING

The user can cause simulated time to advance by typing "SIMULATE C" (simulate for a clock period), "SIMULATE S" (simulate for the STEP time), or "SIMULATE <val>" (simulate for <val> nano-seconds). 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 also be used to immediately cause the evaluation of any zero-delay parts.

7.9 SESSION LOGGING

Creating a permanent record of a simulation session is often useful for future reference. A list file may be created that contains a summary of the directives, errors found in the expansion file, and if requested, a copy of all command inputs. The SNAPSHOT command sends an image of the status lines and signal display window to the list file. For more details, see the section on Logic Simulator Directives.

7.10 INVOKING COMPILER FROM SIMULATOR

Traditionally, the Simulator has learned about the circuit definition by reading in the expansion file and the synonyms file produced by the Compiler. With the advent of ValidSIM, the user can now invoke the Simulator immediately after a design is entered into GED - the Simulator can invoke the Compiler directly and no longer requires the pre-existence of the two Compiler files.

The data obtained by the Simulator is identical to the data that would be obtained if the Compiler were invoked explicitly. The primary difference to the user is the execution time. With ValidPAGECOMP, compilation proceeds more rapidly since only those pages which have changed need to be re-compiled (see Compiler in Chapter 5). The time required for compilation is further reduced since the Compiler need not create its two potentially large files and the Simulator need not read them. Thus, the time required between the creation of a drawing in GED and the start of actual simulation has been reduced significantly.

There are two ways to invoke the Compiler from within the Simulator. One is to use the new directive, ROOT_DRAWING, in the Simulator directives file (see Directives section); this directive is identical to that used by the Compiler. The second is to pass the root drawing name to the Simulator as a command line argument, as can be done with the Compiler:

```
simulate <root drawing name>
```

A command line argument will override any root drawing name in the directives file. The root drawing name should match that in the Compiler directives file, which is still required when the Compiler is invoked by the Simulator. When the root drawing name is specified in either manner, the COMPILER_OUTPUT and SYNONYM_FILE directives should not be used; even if specified, the two files will be ignored. If the root drawing name is not specified, the traditional expansion and synonyms files must exist, and the Simulator will operate identically to that prior to ValidSIM.

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. Since the design may be divided into separate pages, the error messages may be distributed among many files. These messages are collected and output on the screen when the Compiler has completed, except when the Simulator is invoked under GED. The program COMPERR can also be invoked explicitly to collect all the Compiler error messages.

7.11 SIMULATOR GRAPHICS CAPABILITIES

Graphics capabilities are available in the Logic Simulator either when it's run under the Graphics Editor, or when it's invoked with the terminal type set to GCLUSTER (for graphics cluster). The former is referred to as the split-screen Simulator, and the latter as the graphics Simulator.

Certain capabilities are available in either mode of operation. These include the following:

- Waveforms are displayed using graphics rather than with terminal character representations. See the Waveform Representations section.

- A command menu is displayed along the right-hand side of the screen. Under GED, this menu contains nine entries; in the graphics Simulator, the menu size will vary with the window size, with between nine and 17 entries.

- The puck is enabled and available for use with the menu and various commands. A puck point can often be entered in place of typing a signal name. See the Command Summary for a description of puck use with commands.

- The HARDCOPY command can be invoked to produce hard output. See the description of HARDCOPY in the Command Summary.

These two modes also share a limitation - when running under UNIX, the job control feature (ctrl-Z) cannot be used to stop the Simulator. Attempts to stop the Simulator will simply be ignored. Job control capability is enabled with any other terminal type.

There are differences between the graphics capabilities provided with each mode and between these modes and normal terminal operation. These are described in greater detail below.

The graphics capabilities available with the graphics Simulator are not available on the SCALD System IV. However, a graphical character set is used to display waveforms on this terminal (see Waveform Representations section). The user does not need to issue any special commands to use this character set, and waveforms appear more realistic than using normal ASCII characters.

SPLIT-SCREEN GED/SIMULATOR OPERATION

The Graphics Editor SIMULATE command creates a Simulator window and invokes the Simulator. A sufficiently large window is required to run the split-screen Simulator; this minimum size is 48 x 86 characters. The user is prevented from invoking a Simulator in a GED window which is smaller than this minimum. Any window larger than this, up to and including full-screen windows, can be used to run the split-screen Simulator.

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. When invoked, the Simulator does its normal initializations and initializes its window appropriately.

Differences from Normal Terminal Operation

- The user may specify signals visible in the upper (GED) window by pointing to them with the puck instead of typing them at the keyboard. For example, to open a signal, touch OPEN in the menu with the puck, and then point to the signal to be opened. The user can open an unnamed signal by pointing 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 EXIT command terminates the Simulator and causes the Simulator window to disappear.

- 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, the Simulator must be EXITed, the drawing must be recompiled, and the Simulator must be restarted. It is not necessary to exit the Graphics Editor.

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

GRAPHICS SIMULATOR OPERATION

The graphics Simulator is available on an S-32 terminal when the terminal type is set to GCLUSTER. This mode of operation makes available functionality which was previously available only when running the Simulator under GED.

A sufficiently large window is required to run the graphics Simulator; this minimum size is 14 x 86 characters. The user is prevented from running the graphics Simulator in a window which is smaller than this minimum. Any window larger than this, up to and including full-screen windows, can be used to run the graphics Simulator.

There are important differences between this mode of operation and that available when running under GED:

- Up to 48 lines of graphical waveforms can be displayed simultaneously when running the graphics Simulator, as compared to the 12 available under GED.

- The speed of graphical output is significantly faster than that which is possible when running under GED. Output can be made at a speed approximately five times greater.

- The user can control the window size in which the Simulator is invoked and, hence, the number of waveforms which are to be displayed.

- Additional menu commands are available when the Simulator is run in a sufficiently large window. This was described above.

- Additional options are available with the HARDCOPY command. Further details are available in the description of HARDCOPY.

- Communications with GED in another window has not yet been implemented in the graphics Simulator, so it is not yet possible to select signals from GED. However, it is still possible to specify signals by pointing to them in the Simulator display area.

7.12 COMMANDS USED IN WAVEFORMS MODE

The following commands are commonly used in the WAVEFORMS mode and may affect the signal display:

Waveforms { <start time> { <end time> | ; } | ; }
or
Waveforms <point1> { <point2> | ; }

This command invokes WAVEFORMS mode, and defines the range of time to be displayed. If the <end time> parameter (or <point2>) is omitted, then the current display width (<end time> - <start time>) is used with a new <start time>. The WAVEFORMS command may be issued any time the Simulator displays a "*" prompt. Since the WAVEFORMS command does not affect the state of the recorded history of open signals, pan and zoom can be used.

The second syntax is only available when the puck is enabled in the Simulator and is used to zoom in on an area displayed on the screen. <point1> and <point2> are points supplied by the puck, where the left one refers to the <start time> and the right one refers to the <end time>.

The <start time> and <end time> fields may be specified in either absolute or in relative time. Absolute time is specified in nanoseconds, while relative time is specified as RIGHT <offset in nanoseconds> or LEFT <offset in nanoseconds>. If time is specified in relative form, then the new value is calculated by adding (RIGHT) or subtracting (LEFT) the offset from a command dependent value.

For the WAVEFORMS command, the <start time> parameter is relative to the old <start time>, and the <end time> parameter is relative to the new <start time> so that the command

WAVEFORMS RIGHT 10 RIGHT 100 ;

moves the left side of the display to the right by 10 nanoseconds in the history of the open signals, and sets the display width to 100 nanoseconds. The WAVEFORMS display advances automatically whenever simulation is done.

History { <recording period> | ; }

The History command sets or reports the recording period during which a signal history is maintained. If a <recording period> is entered, the Simulator maintains the behavior of all signals opened in WAVEFORMS mode for the specified number of ticks (in the same units as the display). If no parameter is given, then the current value of the recording period is displayed. This value is initially set to 10000 ns.

Cursor <new time> [;]

This command moves the cursor to a new time. The <new time> parameter may be specified in absolute time or relative to the current cursor time. The command

CURSOR LEFT 25

moves the cursor 25 ns to the left. 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. When simulation is complete, the cursor is automatically moved to the current time.

Open <signal name> [, <row> [, <col>]] [;]
or
Open <signal pt> [(<dest pt> <signal pt>)...] [<dest pt>] ;

OPENing a signal adds that signal to the display.

The second syntax is only available when the puck is enabled in the Simulator (GED or GCLUSTER); <signal pt> is a point supplied by the puck that refers to a signal in the drawing or in the Simulator window, and <dest pt> is a point supplied by the puck that refers to the place to display the signal. If <dest pt> is omitted, the Simulator opens the signal in a default location. The sequence of (<dest pt> <signal pt>) can be arbitrarily repeated many times, and must be terminated by a semicolon or carriage return. In WAVEFORMS mode, OPENing a signal also causes its history to be recorded.

When a signal has not already been opened and empty rows are present on the current screen, if the user omits <row>, the signal appears 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. 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. This feature allows a user to OPEN more signals than can be displayed at once, SIMulate to calculate their behavior, and then view their behavior.

DELta_time <point1> <point2>

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 where puck usage is enabled. The value will be returned in the echo area. The points can be specified anywhere in the waveforms display area within the time frame currently being displayed (i.e., valid points are any place where waveforms can be drawn).

ROw <top row number> [;]

This command controls which signals are displayed on the screen. The <top row number> parameter designates which row is to be placed at the top of the screen and may be specified as an absolute value (the top row for the screen) or as an offset relative to the current top row number (by preceding the number with a "+" or "-"). The number of the top row currently displayed on the screen is indicated on the status line as "Top Row". 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 [ON | OFF] [;]

This command allows the user to control the automatic scrolling feature of the Simulator. The Simulator will normally cause the display to scroll in WAVEFORMS mode when a signal not currently on the screen is OPENed. Using this command to turn the feature OFF allows the user to OPEN and DEPOSIT into signals that are not on the display.

7.13 BREAKPOINTS

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).

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 from a command file, the normal breakpoint message is displayed and execution from the script continues. 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 if REALFAST is operating.

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 <expression> [;]
Sets <signal> to 1 when <expression> 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 <expression> [;]
Clears <signal> to 0 when <expression> 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 <expression 1> WHEN <expression 2> [;]
Equates <signal> to the value of <expression 1> when <expression 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 <expression 2> [;]
Equates <signal> to the value of <expression 1> when <expression 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 <expression> [;]
Continuously gives <signal> the value of <expression>. The signal is a "new" signal that is created the first time it is referenced and cannot already exist in the design.

SEt Breakpoint <expression> [;]
Installs <expression> 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 on an IBM host, the # sign prefix must be replaced by the % symbol).

CLear Breakpoint <signal> [;]
Deactivates the signal as a breakpoint. The breakpoint no longer affects simulation (the breakpoint remains in the breakpoint list, but is marked "inactive"). Only simple breakpoints (i.e., breakpoints that are named signals) may be cleared by name; all others must be cleared by number.

CLear Breakpoint # <number> [;]
Deactivates the indicated breakpoint. The breakpoint no longer affects simulation (the breakpoint remains in the breakpoint list, but is marked "inactive").

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.

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.

EXPRESSION SYNTAX

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

```
<expression> -> <expression> OR <boolean expression>  { boolean OR }
             -> <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>       -> <factor>

<factor>     -> <signal>
             -> ( <expression> )
             -> NOT <factor>     { 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 <term>s by spaces to prevent confusion in parsing. Some useful examples are:

```
SET BREAKPOINT NOT BAR
SET BREAKPOINT FOO<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) ;
```

7.14 LOGIC PATCHING

The Simulator has a logic patching facility that 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 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 <expression> [;]
Sets <signal> to 1 when <expression> is true. The signal must be present in the design.

CLear Patch <signal> WHEN <expression> [;]
Clears <signal> to 0 when <function> is true. The signal must be present in the design.

SAmple Patch <signal> GETS <expression 1> WHEN <expression 2> [;]
Equates <signal> to the value of <expression 1> when <expression 2> changes from a 0 to a 1. The signal must be present in the design.

LAtch Patch <signal> GETS <expression 1> WHEN <expression 2> [;]
Equates <signal> to the value of <expression 1> when <expression 2> is a 1. The signal must be present in the design.

EQuate Patch <signal> TO <expression> [;]
Continuously gives <signal> the value of <expression>. The signal must be present in the design.

List Patches [;]
Lists all PATCH signals that have been defined and their definitions.

7.15 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 mode 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.

REQUIRED INFORMATION 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 FOR STANDARD TRACING

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 STANDARD 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 STANDARD 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 will be supported as both a binary and as an ASCII file of signed decimal integers.

FILE FORMAT FOR TABULAR I/O

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. For example:

```
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,10010111;
         30 / 0,5,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 "," 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 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,Z1010112,10, ... 101,UUUUU,1,11,1,111,~
1,11,0101011,1,1, ... 001,1,1,1,1,1,1,1,10,~
010100,
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ,1,11
20 / 0,1,0010101, ... etc.
```

In this case, a new line was inserted before the ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ value 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, at present, memories cannot be traced using Tabular I/O.

SAMPLE TABULAR I/O USE

With the directive "TABULARTRACE 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 foo               { specify signals to trace }
*   .
*   .
*   .
* trace bar<50..3>
* trace_start             { open the file, start tracing }
* sim 100                 { OPEN signals,    }
*   .                     {  deposit values, }
*   .                     {  advance time,   }
*   .                     {  etc.            }
* 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 might execute the following:

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

If stimulus from more than one tabular input file is desired, the TRACE_RESET command can be used to reset the Simulator. For further details, see the description of the TRACE_RESET command.

Logic Simulator Command Summary

7.16 SIMULATOR COMMANDS

All 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 CAPITAL letters.) All command inputs may be typed in either upper or lower case. Some commands prompt for arguments if none are given. The following paragraphs describe the individual commands.

Assertions <signal>, <timing data> [;]

The ASSERTIONS command allows timing assertions to be specified while running the Simulator. This allows the user to specify assertions interactively rather than with the signal name given when creating the drawing in GED. This feature provides the user with an extra degree of flexibility when performing simulations since signal timing assertions are no longer fixed with the signal name and need not be compiled with the drawing.

The < timing data > parameter is specified using the standard SCALD syntax for timing assertion data (e.g., 0-4). The assertion type should not be specified - the Simulator automatically adds the "!C" property to the timing data. This command can be invoked on existing clock signals as well as any other signals in the drawing. Thus, any signal can be assigned timing assertions while in the Simulator, and assertions of existing clock signals can be re-defined. After assigning clock properties, the signal can be OPENed using either its previous or its new (with assertions) name.

Bus [;]

Places the user in BUS mode and refreshes the screen.

CLear Breakpoint <signal> [;]

Deactivates a named breakpoint. See Breakpoints section.

CLear Breakpoint # <number> [;]

Deactivates a numbered breakpoint. See Breakpoints section.

CLear Enable <signal> WHEN <expression> [;]
Clears an ENABLE signal when <expression> is true. See Breakpoints section.

CLear Patch <signal> WHEN <expression> [;]
Clears a PATCH signal when <expression> is true. See Logic Patching section.

CLOck [ON | OFF] [;]
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 <val> [;]
Compares the value of the currently open signal against <val>. 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 will not track the number of transitions.

Cursor <new time> [;]
Moves the WAVEFORMS cursor to a new time. See WAVEFORMS section.

DELta_time <point1> <point2>

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 where puck usage is enabled (GED or GCLUSTER). See WAVEFORMS section.

Deposit [<signal>,] <val> [;]

Deposits <val> into the indicated <signal>. Multiple bit values appear in the current radix. <signal> is an optional parameter which may be specified using the puck. If <signal> is not specified, <val> will be DEPOSITed on the currently OPEN signal. Note that this command will neither OPEN the specified signal nor change which signal is currently OPEN. If the specified <signal name> has not been OPENed, DEPOSIT will cause the value to be placed on the signal, but will neither OPEN it nor cause signal history to be started.

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 will indicate this to the user. The output to the echo area in response to the commands given will proceed as usual. When the display is reenabled, the screen will be 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; 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.

EQuate Enable <signal> TO <expression> [;]
Equates an ENABLE signal to <expression>. See Breakpoints section.

EQuate Patch <signal> TO <expression> [;]
Equates a PATCH signal to <expression>. See 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 terminal type GCLUSTER. 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 section on WAVEFORMS.

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 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 <val> [;]
Sets the number of clock intervals to the specified decimal integer <val>. The interval value appears on the status lines. 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 Breakpoints section.

LAtch Patch <signal> GETS <expression 1> WHEN <expression 2> [;]
Latches a PATCH signal to <expression 1> when <expression 2> is true. See Logic Patching section.

List Breakpoints [;]
Lists all breakpoints. See Breakpoints section.

List Enables [;]
Lists all ENABLE signals. See Breakpoints section.

List Patches [;]
Lists all PATCH signals. See 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 would appear:

```
FOO,binary
BAR<66..33> ,hex
BAR<4> ,binary
```

LOadmemory (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 decimal regardless of the current radix. The MEMLOAD command is discussed in detail later in this chapter.

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 GCLUSTER).

Nextmemory [;]

Advances the mempath to another memory.

Open <signal name> [, <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 using the puck with the Simulator running under GED. <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 arbitrarily 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 If a location is specified, the memory display is placed at that location (if possible).

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 <val> [;]

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

PLot [<starting_time> <ending_time>] ['<filename>'] [;]
Builds a timing diagrams file for plotting via the PLOTTIME program and 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; after the initial invocation, subsequent calls without the <filename> parameter append additional data onto the previously specified file. For additional information on the Plottime program, see "Plottime Timing Diagram Program" in Chapter 6.

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. Previously, a signal had to be OPENed in WAVEFORMS mode in order to start a recording of its history. Thus, after a period of simulation, if a signal was not OPENed, there would be no method to determine what the value of a signal was at a previous time. 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 will be REMOVEd. If <signal name> is entered from the keyboard, a single occurrence of the signal in the current radix will be REMOVEd. When selected from the menu, the user will be prompted for a signal name.

RESume [;]
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 WAVEFORMS section.

SAmple Enable <signal> GETS <expression 1> WHEN <expression 2> [;]
Samples an ENABLE signal to <expression 1> when <expression 2> becomes true. See Breakpoint section.

SAmple Patch <signal> GETS <expression 1> WHEN <expression 2> [;]
Samples a PATCH signal to <expression 1> when <expression 2> becomes true. See 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. See the PAUSE and RESUME commands for further information on script files.

SCROll [ON | OFF] [;]
Allows the user to control the automatic scrolling feature of the Simulator. See WAVEFORMS section. The default is ON.

SEt Breakpoint <expression> [;]
Installs <expression> as a breakpoint. See Breakpoints section.

SEt Breakpoint # <number> [;]
Activates a numbered breakpoint. See Breakpoints section.

SEt Enable <signal> WHEN <expression> [;]
Sets an ENABLE signal when <expression> is true. See 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. Spool files have names of the form 'hardXX', where XX is the tty number of the current window. This command is only available with the graphics Simulator - it has no effect when running under GED. LOCAL_PLOT is the default. See the HARDCOPY command.

SEt Patch <signal> WHEN <expression> [;]
Sets a PATCH signal when <expression> is true. See Logic Patching section.

SEt { W11versatec | W22versatec | W36versatec | W42versatec | Calcomp1043 | Calcomp5744 | B9424 } [;]
Specifies the plotter type for HARDCOPY output. 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 developing Logic Simulator models.

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 (matching its former behavior), but any value between 0 and 100, inclusive, 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.

STep <val> [;]
Sets the simulated time step size to the specified decimal integer <val>.

Terminal { Vt100 | Cluster | Gcluster | Annarbor | Tty | 3270 } [;
Sets the terminal type. Accepted types are:

VT100	a DEC VT100 with 24 lines.
CLUSTER	the SCALD CLUSTER terminal in transparent mode connected to the host computer.
GCLUSTER	the SCALD CLUSTER terminal with graphics capabilities enabled.
ANNARBOR	an Ann Arbor Ambassador terminal, 48 lines.
TTY	a video terminal (this is the default).
3270	an IBM 3270.

TRace <signal name>, [<radix>] [;]
or
TRace <point> [<point> ...] [;]
Traces the output or outputs corresponding to the given signal or signal subrange. The second syntax indicates that <signal name> may be specified using the puck to point at it. <radix> is an optional parameter which 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 TRACE_RADIX and LIST TRACES commands for more information.

Example:
```
* trace foo
* trace bar<66..33>,h
* trace bar<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 <file name> [;]
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. To use two (or more) input stimulus files, a sequence of commands similar to the following could be used:

```
* trace_read file1
* sim c
.
. etc.
.
* logic_init -*
* trace_reset
* trace_read file2
* sim c
.
. etc.
.
```

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.

Update_interval <constant> [;]

Sets the simulator to update the screen at specified intervals while simulating for a longer time. If zero (0) is specified, any previously set interval is cleared and updating is disabled.

Waveforms { <start time> { <end time> | ; } | ; } [;]
or
Waveforms <point1> { <point2> | ; }

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

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. Also see the COVERAGE and INIT_COVERAGE commands.

Logic Simulator Directives Summary

7.17 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 Compiler's expansion file. Directives must appear in the Simulator directives file.

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. An example of a Logic Simulator directives file is given at the end of this section.

BINARY_TRACE { ON | OFF } ;

This directive is ignored for Tabular tracing. 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.

CLOCK_ON_DRIVEN { OFF | ON } ;

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 will no longer be allowed unless this directive is specified as ON.

CLOCK_PERIOD integer ;

Sets the period of the clock (in nanoseconds) used by the Simulator. Any signal with a "C" or "P" name property (e.g., MASTER CLK !C 0-3) has its behavior specified relative to this period.

```
CLOCK_PERIOD 56;  { sets the clock period to 56 ns }
```

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

CLOCK_INTERVALS integer ;

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.

```
CLOCK_PERIOD 100;  { sets the clock period to 100 ns }
CLOCK_INTERVALS 20;  { divides the clock into 20 units }
```

Using the above values, the signal MASTER CLK !C 0-10,15-20 is high both from 0 ns to 50ns and from 75 ns to 100 ns. In terms of a Timing Verifier timing description: MASTER CLK !C 0-10,15-20 = 1:0, 0:50, 1:75 . If CLOCK_INTERVALS is unspecified, the clock is divided into 10 sub-periods.

COMPILER_OUTPUT 'filename' ;

Specifies the name of the Compiler output file containing the design to be simulated. If no Compiler output file is specified in the directives file, the default filename 'cmpexp.dat' is used. The file name must be enclosed in quotes.

```
COMPILER_OUTPUT '[jane.qa]cmpexp.dat';
```

COMMAND_FILE 'filename' ;

Specifies the name of a command file to be invoked immediately after the Compiler output file is read. The Simulator can be run in batch mode by means of such a command file. The file name must be enclosed in quotes. See the SCRIPT command for a description of command files.

```
COMMAND_FILE '[jane.qa]comfile.dat';
```

DECAY_TIME time ;

Specifies the time period during which MEMORY strength signals retain their value (i.e., before they assume an UNDEFINED value). The default value is infinite - MOS signal strengths will not decay over time unless the user explicitly specifies a decay time.

```
DECAY_TIME 10000;
```

LOGIC_STATE { 2 | 4 } ;

This directive is no longer supported. All signals assume one of 12 states.

MEM_STATE { 2 | 4 } ;

Selects between two-state memories and four-state memories. A four-state memory retains U's. If not specified, memories are four-state (in fact, a misnomer since there are only three actual states).

```
MEM_STATE 2;
```

OUTPUT [NO] { LIST , COMMAND_LOG } ;

Determines output files produced by the Simulator. If no directive is given, no files are created. The output file specifiers are:

LIST	causes the LSTFILE file to be created. The contents of the list file are controlled by other directives.
COMMAND_LOG	is a file containing only 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 SCRIPT command).

REALCHIP_LIBRARY 'filename' ;

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 present if any Realchip models are used by the Simulator. Otherwise, this directive can be omitted. The file name must be enclosed in quotes.

```
REALCHIP_LIBRARY '[jane.qa]realchip.dat';
```

RESOLUTION time ;

Specifies the time resolution to be used by the Simulator. 'time' is specified as a real number of nanoseconds (<1 for finer resolution, >1 for coarser), the default value is 1 ns. The resolution being used by the Simulator is indicated in the display area as a fixed point value labeled "Scale:".

RESOLUTION 0.05;

This directive affects the user interface in several areas. The time scale in WAVEFORMS mode will no longer represent nanoseconds, but must be scaled by the indicated scale factor; using the above example, each tick (formerly 1 ns) will now represent 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.) will maintain their relation to ticks on the screen, although the "real" times associated with those ticks has changed (e.g., "WAVE 0 1000" will display a time scale of 0 to 1000 ticks, representing 50 ns of time).

Exercise caution when manipulating 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 { ON | OFF } ;

Specifies if separate RISE/FALL delays will be used by the Simulator. If the ON state is specified, simulations will be performed using both the rise and fall delays specified for parts. The default state of this directive is OFF; this causes all primitives to change states after the specified delay time (if only one value is given) or after the greater of the rise and fall delays. See Delays section.

ROOT_DRAWING 'drawing name' ;

Specifies the drawing's name when direct invocation of the Compiler from the Simulator is desired. The traditional expansion file and synonym file are not needed and will not be created when the Compiler is invoked from within the Simulator.

```
ROOT_DRAWING 'counter';
```

SESSION_LOG { ON | OFF } ;

Specifies if a copy of terminal I/O in to be output to the List file. Note that SESSION_LOG ON and OUTPUT NO LIST are incompatible.

```
SESSION_LOG OFF;
```

SIGNAME_CHARS { 9 - 24 };

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. Values outside the legal range will be rounded to the closest legal value.

```
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.

SYNONYM_FILE 'filename' ;

Specifies the name of the synonyms file, which contains the sets of names that each signal is known by. The synonyms file is created by the Compiler. If no synonyms file is specified in the directives file, the default filename 'cmpsyn.dat' is used. The file name must be enclosed in quotes.

```
SYNONYM_FILE '[jane.qa]cmpsyn.dat';
```

TABULAR_TRACE { ON | OFF } ;

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

TERMINAL { VT100 | CLUSTER | GCLUSTER | ANNARBOR | TTY | 3270 } ;

Specifies the terminal type. VT100 is assumed to be a DEC VT100 (or equivalent) with 24 lines. CLUSTER is assumed to be a SCALD CLUSTER terminal running the Simulator locally or in transparent mode connected to the host computer. GCLUSTER is identical to the CLUSTER type except that graphics capabilities are also included. ANNARBOR is assumed to be an Ann Arbor Ambassador terminal with 48 lines. TTY is assumed to be any dumb video terminal. 3270 is assumed to be an IBM 3270 or equivalent.

```
TERMINAL VT100;
```

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

TRACE_RADIX { 2 | 8 | 10 | 16 } ;

Specifies the default radix to use for tabular tracing. The default is initially 2.

```
TRACE_RADIX 16;
```

USE_IF { BATCH | INTERACTIVE } ;

Precedes directives that 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 USE_IF directs 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.

```
USE_IF BATCH;
COMMAND_FILE '[JANE.QA]BATCHSIM.CMD';
TERMINAL TTY;
SESSION_LOG ON;
```

USE_REALFAST { ON | OFF } ;

Controls use of Realfast simulation accelerator. When enabled (USE_REALFAST ON;), a simulation is aborted if the Simulator cannot access the Realfast hardware (Realfast currently is not a shareable resource; simultaneous use by more than one work station is prohibited). Simulation using Realfast is the same as without its use -- Realfast simply increases the speed of simulation. If this directive is omitted, the Realfast simulation accelerator is not used.

USE_SYNONYM { ON | OFF } ;

Determines if the Simulator is required 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_PRIM_CONFIG 'filename' ;

Specifies the name of the user primitive configuration file that contains the pin names of the user-coded primitive in the format explained in the section on User-Coded Simulator Primitives; the filename must be quoted.

```
USER_PRIM_CONFIG '[jane.qa]primconf.dat';
```

WIRE_DELAYS 'filename' ;

Specifies the name of the wire delays file; the filename must be quoted. See Wire Delays section.

```
WIRE_DELAYS '[jane.qa]wiredel.dat';
```

AN EXAMPLE OF A SIMULATOR DIRECTIVES FILE

The Simulator directives file is created with a text editor. The Simulator ignores carriage returns and multiple spaces. Directives may be entered in either upper or lower case. Comments may be included if enclosed in curly brackets. Note that each directive must be terminated with a semicolon (";") and that the file must end with an "END." statement.

```
CLOCK_PERIOD 100;           { sets the clock period to 100 ns          }
CLOCK_INTERVALS 5;          { clock has five intervals of 20 ns        }
OUTPUT LIST;                { creates a list file                      }
SESSION_LOG ON;             { creates a session log                    }
COMPILER_OUTPUT '[JANE.SIM]CMPEXP.DAT';
                            { name of Compiler expansion file          }
SYNONYM_FILE '[JANE.SIM]CMPSYN.DAT';
                            { name of synonym file                     }
TERMINAL CLUSTER;           { terminal type                            }
USE_IF BATCH;               { remainder of directives for batch only   }
COMMAND_FILE '[JANE.SIM]BATCH.CMD';
                            { name of command file                     }
TERMINAL TTY;               { terminal type for batch mode             }
END.                        { marks end of the file, note "."          }
```

7.18 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 and must be enclosed in the indicated brackets ('<>' and '[]' respectively). 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 [mem size-1..0] where mem 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 site-default bit ordering).

7.19 RISE/FALL DELAY PROPERTY

Delay values associated with Simulator primitives may include a rise delay and a fall delay. Specification of these delays is made through the DELAY property or through the properties, RISE and FALL.

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. Thus, delay can be specified in one of the following formats:

```
DELAY <delay time>
DELAY <rise delay>, <fall delay>
```

In addition, 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>
```

Note that the DELAY property and the RISE and FALL properties should not both be specified on the same body or an error will result.

The RISE_FALL directive is used to control the use of separate RISE/FALL delays. The format of this directive is as follows:

```
RISE_FALL { OFF | ON }
```

If the ON state is specified, simulations are performed using both the rise and fall delays specified for parts. The default state of this directive is OFF, which causes all primitives to change states after the specified delay time (if only one value is given) or after the greater of the rise and fall delays.

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

old value	new value	delay to use
X	0	fall
X	1	rise
X	U	min(rise,fall)
0	Z	rise
1	Z	fall
U	Z	max(rise,fall)

7.20 WIRE DELAY FEEDBACK

Wire delays can be fed back in either of two ways:

1. By using a directive of the form

```
WIRE_DELAYS 'filename';
```

2. By using a command of the form

```
WIRE_DELAYS filename [;]
```

The file must be in the format described below. Basically, each element 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 bit. These delays are added in with any other specified delay values to determine when Simulator events should be scheduled for those bits.

```
<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 range> |
                          <quoted path name> =
                               <quoted rise/fall range>

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

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

<bit number>          ::= <integer>

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

```
<rise delay range>  ::= <min delay> - <max delay>

<fall delay range>  ::= <min delay> - <max delay>

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

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

<delay range>       ::= <delay>, <delay> |
                        <delay> - <delay>

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

At present, the Simulator does not support the following:

```
<stop delay>        ::= <quoted path name> =
                                <quoted rise/fall range>

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

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

<delay range>       ::= <delay> - <delay>
```

In other words, the delay specified for a signal is applied to all of its inputs. Note that if only <rise delay range> or only <fall delay range> is specified, the maximum delay is applied.

The following is an example of a wire delay file:

```
'DATA' <5..0>: = '2.3, 3.4';

'ENABLE' : = '5.1';

END.
```

7.21 A SAMPLE SIMULATION SESSION

First, use the Graphics Editor to create the circuit to be simulated. The Simulator can then be invoked immediately to process the design (in turn invoking the Compiler), or the Compiler can be explicitly invoked to create data files for the Simulator. The former method is more efficient since the generation and processing of Compiler data files is eliminated.

COMPILATION BEFORE SIMULATION

Compile the design for simulation (i.e., compile it using the Compiler directive "COMPILE SIM"). The Compiler directive, DIRECTORY, specifies the SCALD directories read by the Compiler. The directories for all drawings referenced by the root drawing, including the drawings for every part used, the special parts like "not bodies" and "B size pages," and the Simulator primitives, must be included in the directives file.

The following is a sample Compiler directives file:

```
root_drawing 'joes circuit';
compile sim;
directory '/u0/lib/standard/standard.lib',
          '/u0/lib/sim/sim.lib',
          '/u0/lib/lsttl/lsttl.lib',
          '/u0/joe/joe.wrk';
warnings on;
oversights on;
output list, expand;
print_width 80;
end.
```

In the sample directives file, the drawing to be compiled is named "joes circuit." This drawing resides in the SCALD directory "joe.wrk". Since this drawing includes both LSTTL parts and Standard parts (Merge bodies, Not bodies, B size pages, etc.), the LSTTL and Standard libraries (directories) are included; the Sim library, which contains the Simulator primitives, must be specified. Note that any errors reported during compilation must be corrected before the design is simulated.

The Simulator is invoked by typing the word "simulate" in response to the shell prompt. The Simulator reads its directives file (see Directives Summary) and checks the file for correctness. It then reads the Compiler's expansion and synonyms files and constructs its internal representation of the circuit. If no errors are found, simulation is begun; the user may open signals, advance simulated time, and execute any

desired Simulator commands (see Command Summary) to observe and test the circuit. When the user has completed simulation, exit the Simulator by entering the command "exit."

COMPILATION WITHIN SIMULATOR

Include the ROOT_DRAWING directive in the Simulator directives file. The specified root_drawing should be identical to that in the Compiler directives file, which is also required. The format of the Compiler directives file is the same whether invoked directly or from within the Simulator (see example above).

Type the word "simulate" in response to the shell prompt. The Simulator reads its directives file (see Directives Summary) and checks the file for correctness. When the ROOT_DRAWING directive is used, the COMPILER_OUTPUT and SYNONYM_FILE directives should not be used; even if specified, any existing expansion and synonyms files will be ignored. The Simulator will then invoke the Compiler and receive the circuit description from the Compiler directly; the Compiler will NOT generate the expansion and synonyms files. Note that any errors reported during compilation must be corrected before the design is simulated.

If no errors are found, simulation is begun; the user may open signals, advance simulated time, and execute any desired Simulator commands (see Command Summary) to observe and test the circuit. When the user has completed simulation, exit the Simulator by entering the command "exit."

7.22 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 SCALD III 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.

THE PASCAL CODE FOR VAX HOSTS

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;

CONST
  {user constant definitions}
  MAX_PIN_BIT_NUMBER = <value of user's choice>;

TYPE
  {user type definitions}

  %INCLUDE 'SYS$SCALD:USERPRIM.TYP'
  %INCLUDE 'SYS$SCALD:USERPRIM.DCL'
  PROCEDURE userprim;
          .
          .
          .
  END {of procedure userprim};

END {of module}.
```

To compile and link "userprim.pas" with the Simulator on the VAX, 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 on the S-32 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.

THE PASCAL CODE FOR IBM HOSTS

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 user's 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 on the 370, 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 on the S-32 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.

THE PASCAL CODE FOR S-32 HOSTS

The user must code his primitive as a single PASCAL procedure that is 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 on the S-32, 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.

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 on the different hosts.

Running Your Simulator on the S-32

To run your Simulator under the Graphics Editor on the S-32, start the Graphics Editor as you normally would, and EDIT your drawing. When you want to run the Simulator, type:

set user_sim <name_of_your_simulator>
simulate

The Simulator name must be specified with its full pathname. The Simulator specified will be invoked.

To run your Simulator without GED, 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 on the VAX

Type:

```
@SYS$SCALD:SIMASSIGN
RUN SIM
```

where SIM is the name of your version of the Simulator.

Running Your Simulator on the 370

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.

BODY DEFINITION FOR UCPs

The body definition of a UCP is nearly the same as the body definition for any other primitive part -- see the guidelines outlined in Valid Library Styles and Standards located in Chapter 11. 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

```
FILE_TYPE = SIM_DIR;
```

as the first line.

Since SCALD directories created by GED have

```
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.

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 '<primitive 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 '<primitive 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.
```

- 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 names> and <primitive names> are strings containing no "'" and no ':'. A <primitive name> must be no longer than 20 characters. There is no restriction on the length of a <pin name>. The pin name should be in canonical syntax. For example, the name '-G' is the canonical syntax for 'G*'.

- <size> specifies how wide the pin is:

  - <size> = SIZE if the pin is to have the subscript
    <size-1 .. 0> (right_to_left bit ordering) or
    <0 .. size-1> (left_to_right bit ordering)

  - <size> = an integer K if the pin is to have the subscript
    <K-1 .. 0> (right_to_left bit ordering) or
    <0 .. K-1> (left_to_right bit ordering)

  If K=1, then the pin is interpreted as SCALAR (i.e., no subscript).

- 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:

```
USER_PRIM_CONFIG <file_name>;
```

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..<user's spec>] OF INTEGER;
```

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_pin_bit_number]
                  of logic_type;
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 greater, 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).

```
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.

7.23 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 the names of the primitives, the names and widths of the input and output pins, the amount of user storage required for each instance of the primitives, 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.
```

USER'S VAX PASCAL MODULE EXAMPLE

```
(*
*)
(*$S-,C+,X-,W-*)
MODULE USERPRIM;

(***************************** 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.                                           *
        *                                                       *
        *********************************************************)

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[0],s2) and
          logic_to_int(s[1],s1) and
          logic_to_int(s[2],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 := u_size-1 downto 0  do         (* do function *)
    case select of

  0: f[i] := logic_0;    (* CLEAR *)

  1: begin
        (* later *)
     end;

  2: begin
        (* later *)
     end;

  3: begin       (* PLUS *)
       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_PRIM_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.
(*
*)
```

7.24 SIMULATOR MODELING

Every part used in a circuit must be modeled in terms of Logic Simulator primitives if a simulation is to be performed on that design. A wide variety of Logic Simulator primitives are available from simple logic gates to a complete ALU. The behavior of each primitive is understood by the Logic Simulator.

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 this appears as follows:

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 would be used, not 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 some easily discernible abbreviation.

7.25 SIMULATOR PRIMITIVES

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

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.

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.

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

The OR primitives also come in seven varieties: 2 OR, 3 OR, 4 OR, 5 OR, 6 OR, 7 OR, and 8 OR.

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

The XOR has only a 2-input version.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0
any	U or Z	U
Z or U	any	U

Buffer Primitives

There are three buffer primitives: the simple buffer BUF, the tri-state buffer TS BUF, and the identity buffer, IDENTITY. To create an inverting buffer, simply bubble the input or output pin of a buffer. Non-inverting buffers are commonly used for delays.

The BUF primitive behaves as follows:

Input	Output
0	0
1	1
Z or U	U

The tri-state buffer has an enable input which, when disabled, causes the output to take the value "Z." The enable input has a width of one bit.

Input	Enable*	Output
0	0	0
1	0	1
Z or U	0	U
any	1	Z
any	Z or U	U

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.

Input	Output
0	0
1	1
Z	Z
U	U

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.

J	K	Clock	PR*	CL*	Q	Q-BAR
any	any	any	Z or U	any	U	U
any	any	any	any	Z or U	U	U
any	any	any	0	0	1	1
any	any	any	0	1	1	0
any	any	any	1	0	0	1
any	any	any→Z,U	1	1	U	U
any	any	any→0	1	1	no change	no change
0	0	0→1	1	1	no change	no change
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

Latch Primitives

There are three latch primitives: the LATCH, LATCH RS, and LATCH RS COMP. The latches 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 LATCH primitive behaves as follows:

Data	Enable	Output
any	0	no change
0	1	0
1	1	1
Z or U	1	U
any	Z or U	U

In the LATCH RS primitive, reset prevails over set if both are asserted.

Data	Enable	PR*	CL*	Output
any	any	any	Z or U	U
any	any	any	0	0
any	any	Z or U	1	U
any	any	0	1	1
any	Z or U	1	1	U
any	0	1	1	no change
0	1	1	1	0
1	1	1	1	1
Z or U	1	1	1	U

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

Data	Enable	PR*	CL*	Output	Output*
any	any	any	Z or U	U	U
any	any	Z or U	any	U	U
any	any	0	0	1	1
any	any	0	1	1	0
any	any	1	0	0	1
any	Z or U	1	1	U	U
any	0	1	1	no change	no change
0	1	1	1	0	1
1	1	1	1	1	0
Z or U	1	1	1	U	U

Register Primitives

There are four register primitives: the REG, REG RS, REG RS COMP, and REG CKE. The registers have an edge-sensitive clock input. 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. The REG RS and REG RS COMP also have asynchronous set and reset inputs that cause the outputs to take the values 1 and 0, respectively.

The REG primitive behaves as follows:

Data	Clock	Output
0	0→1	0
1	0→1	1
Z or U	0→1	U
any	1→0	no change
any	any→Z,U	U

In the REG RS primitive, reset prevails over set if both are asserted.

Data	Clock	PR*	CL*	Output
any	any	any	Z or U	U
any	any	any	0	0
any	any	Z or U	1	U
any	any	0	1	1
any	Z or U	1	1	U
0	0→1	1	1	0
1	0→1	1	1	1
Z or U	0→1	1	1	U
any	1→0	1	1	no change

Complementary outputs are provided on the REG RS COMP, and both outputs take the value 1 when both set and reset are asserted.

Data	Clock	PR*	CL*	Output	Output*
any	any	any	Z or U	U	U
any	any	Z or U	any	U	U
any	any	0	0	1	1
any	any	0	1	1	0
any	any	1	0	0	1
any	any→Z,U	1	1	U	U
any	any→0	1	1	no change	no change
0	0→1	1	1	0	1
1	0→1	1	1	1	0
Z or U	0→1	1	1	U	U

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

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. Clever use of a multiplexer can often drastically reduce the number of Logic Simulator primitives needed to model a part. The table for the 2 MUX is as follows and can be extended readily for the 4 MUX and 8 MUX:

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

Memory Primitive

There is one memory primitive: the MEMORY primitive. 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 write enable and chip select inputs on the VALID-supplied body definitions are bubbled (many actual parts have these inputs low asserted). although they may be un-bubbled if necessary. The Master Reset input, when asserted, clears the entire MEMORY to zeros.

Memories may be modeled in either 2-state or 4-state mode. In 2-state mode, each bit of the memory may assume one of two states: 0 and 1. In 4-state mode, each bit of the memory may assume one of three states: 0, 1 or U.

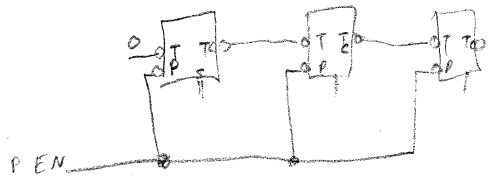
Counter/Shift Register Primitive

An Up-Down counter with right and left shifting capabilities is available: the COUNTER SHIFT REGISTER primitive. This primitive has seven inputs: MR - Master reset, CK - clock, CEP - count enable parallel input (active low), CET - count enable trickle input (active low) that also acts as a serial input for shift left, S - select inputs (3 bits), DI - parallel data in, and MSBIN - serial data input for shift right; it produces two outputs: DO - data out and TC - terminal count (active low).

CET CEP : NO EFFECT ON SHIFTING OR LOADING

CET has combinational path to TC

S " " " 7-90



The function 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.

Arithmetic Primitives

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.

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:

```
 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 denotes 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.

The lookahead carry generator primitive LOOKAHEAD has three inputs - P, G, and CARRY IN - and 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.

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.

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.

Other Primitives

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.

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 (111b).

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.

The PARITY primitive's I input can be sized and produces a one bit output T. T is the ~~parity~~ exor -- exclusive or of I.

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

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.

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 which controls whether the A and B pins are connected; however, since the transistor described is now uni-directional, the A pin is an input pin rather than an output.

7.26 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 Simulator Primitives."

7.27 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 Logic 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.

7.28 SIMULATOR ERROR MESSAGES

Whenever the Simulator encounters an error in input, the Simulator prints the input line along with a pointer to the position in the line where the problem is detected in the simlst.dat file.

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 (=) 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 subrange specifier

Generated when the Simulator is expecting a subrange specifier (..) 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 parenthesis, e.g., (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: Expected (

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

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: Reserved

ERROR #19: Reserved

ERROR #20: Unmatched closing comment symbol

Generated when the Simulator encounters a closing comment symbol ("}") without a matching starting comment symbol ("{"). 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.

ERRORS #26 through #29: Reserved

ERROR #30: Unexpected symbol in bit subscript

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

1. A subrange symbol (..).
2. A colon (:) specifying a bit step.
3. A greater than symbol (>).

ERROR #31: Reserved

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: Reserved

ERROR #36: Constant is too long

ERROR #37: Expected .

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

ERROR #38: Reserved

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: Reserved

ERROR #47: Reserved

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: Can't 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: This file name was already specified

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: Reserved

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: Extraneous characters at end of expr

Generated when the Simulator finds extra characters at the end of an expression. All characters in a string that are to be evaluated as an expression must be a part of the expression.

ERROR #59: Reserved

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 outside the range 2-16. The Simulator supports four radices in that range (2, 8, 10, 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: Reserved

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 14x86

Generated when the graphics Simulator is invoked in a window which is smaller than the minimum size of 14 x 86. Create a larger window for the Simulator.

ERROR #69: Unrecognizable format specification

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.

ERROR #77: Symbol must be one character

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

ERROR #78: This symbol cannot be used here

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

ERROR #79: Subrange symbol must be .. or :

Generated when the Simulator finds a subrange specifier that is neither ".." nor ":".

ERROR #80: No pins found on part in drawing

ERROR #81: No pins found on library part

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 delays 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: Vector PIN_NUMBER < pin's width

ERROR #91: Vector PIN_NUMBER > pin's width

ERROR #92: Invalid error number (warnings only)

ERROR #93: Expected directory file name

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 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. Specify a clock period greater than zero.

ERROR #112: Run stopped because errors were detected

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. The valid terminals are:

1. VT100
2. ANNARBOR
3. CLUSTER
4. 3270
5. TTY

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 memory 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 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: MEM_BLOCK will not fit into memory

Generated when the Simulator is reading a MEMLOAD file and finds that the current mem_block is larger than the memory primitive being loaded. Use the word range specification option of the MEMLOAD command.

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 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, OUTPUT_SPEC, or IO_SPEC symbols 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 in the PERIOD command or CLOCK_PERIOD directive.

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 (: or , or ;) 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 should seldom occur.

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 the Simulator does not find an output type for an output pin. This error seldom occurs because the default is (TS,TS).

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 a RESET_SEQ pin in Realchip.

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 does not find a CLOCK_PIN in the Realchip adapter.

ERROR #154: Realchip adapter not found

Generated when Realchip 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 #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.

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: **** Realchip in use or not present

Generated when a user attempts to use Realchip on a system where it does not exist or is already in use.

ERROR #177: unknown value input to micro sim device

ERROR #178: Feedback may be disconnected

Generated when the feedback connection for a device that requires feedback is disconnected.

ERROR #179: WORD_RANGE does not match memory width

Generated when the optional parameters used in the MEMLOAD command have unequal file and primitive word ranges.

ERROR #180: Static_forever device is not unique

Generated when there is more than one static forever device having the same primitive name in the same design.

ERROR #181: Reserved

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.

ERRORS #185 through #196: Reserved

ERROR #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.

ERROR #198: Inconsistent MEMLOAD parameters

Generated when the Simulator finds a different number of words in the MEMLOAD file from that specified by the optional parameters for the MEMLOAD command.

ERROR #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.

ERROR #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 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

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 indicates a likely hardware failure. If this error occurs, call your field service representative.

ERROR #215: Bad Realfast access

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: Reserved

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 micro-sim pins 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 #230: Reserved

ERROR #231: Expected SPECIAL

Generated when a Realchip part requires SPECIAL sampling and the Simulator finds some other data in the definition file.

ERRORS #232 through #250: Reserved