ValidTIME™ REFERENCE MANUAL

Manual Number: MN225 Rev.A

10 April 1986

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

MANUAL REVISION HISTORY

Rev	Date	Software Release	Reason for Change
A	4-10-86	ValidTIME Release 1.0	Initial release.

TABLE OF CONTENTS

SECTION 1
OVERVIEW

ValidTIME, the SCALDsystem Timing Verifier is a powerful design tool that provides thorough verification of synchronous circuits and can be used early in the design cycle. Because ValidTIME is not dependent on full circuit simulation, accurate timing data can be obtained on portions of a design before a full logic simulation is possible. The two significant advantages to this design methodology are first, that timing information can be incorporated into the design process incrementally, thus assuring more accurate design, and second, that the unbundling of timing verification from logic simulation produces a faster, more efficient, verification tool.

1.1 WHAT IS TIMING VERIFICATION?

Digital systems are composed of components and their interconnections, or wires which convey signals from one component to another. In general, when a signal on the input of a component changes, some time later the signal on the output may change. The wire connected to this output then conveys the signal to the input of other components, again after some delay. Because of variations in construction, the delay time of components and wires varies.

At certain places in a system - data inputs to registers, and external interfaces for example - a signal must assume its value at a certain time. If the delay to such a place is too long or too short, the signal may change value too early or too late. This will cause the system to yield an incorrect result. Consider the D register shown in Figure 1-1:

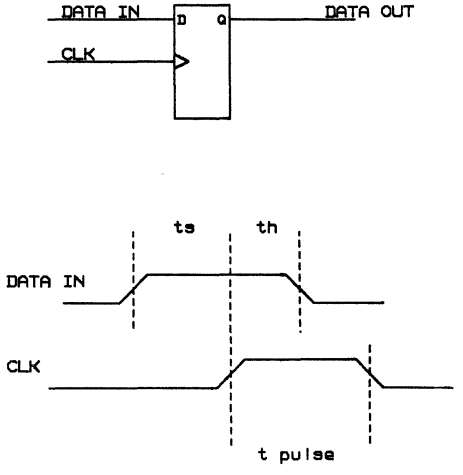

Figure 1-1. D Register

In order for this register to function correctly, the signal DATA_IN must be stable for a specified amount of time before the clock signal CLK changes. This is the setup time for the register and is marked as **ts** in the example. The signal DATA_IN must also remain stable for a specified amount of time after CLK changes. This is the hold time for the register and is marked as **th** in the example.

For the register to function correctly it is also necessary for CLK to remain high for a certain minimum time. The time that CLK remains high is called the clock pulse, and so this requirement is called a minimum pulse width. It is marked in the example as **t pulse.**

When these requirements are met, the signal DATA_OUT will change after the register is clocked. The time interval

between the time CLK changes and the time DATA_OUT changes is the time it takes the DATA_IN signal to propagate through the circuitry of the register. This is called propagation delay. The timing requirements and propagation delays of devices are listed in manufacturer's data books.

As a second example, consider a memory interface comprising a data bus named D<7 . . 0> and a data ready signal named READY, there will typically be some specification that the bits D <7 . . 0> must be ready (that is stable, not changing) some period of time before READY becomes true. If this setup time is not met, systems connected to the memory may malfunction.

The Timing Verifier checks all of the components in the design for their timing constraints by using a timing model for each component. These timing models are supplied in Valid Libraries. They are built by Library Development Engineers on the basis of manufacturer's data books. The Timing Verifier augments the information in the models with user-specified timing information (wire delays, signal assertions, etc.) and checks that user-specified assertions are met.

1.2 TIMING VERIFIER PRINCIPLES

The underlying principle of the SCALDsystem Timing Verifier is that most timing errors occur when signals are changing states, not when they are stable. Therefore only the behavior of changing signals needs to be examined for possible timing errors. When a given signal is stable, whether it is high or low (1 or 0) is of no importance. And when a given signal is changing, whether it is changing from high to low, or low to high, is only important if the rise and fall delays at that particular place are asymmetrical.

By defining the cyclical clock behavior and setting most other signals in a circuit to a stable condition at the outset

of the verification, the SCALDsystem Timing Verifier can test each signal and library part for all possible timing errors during a complete clock period. This thorough timing analysis provides considerably more information than evaluation for worst, best, and typical cases.

A basic assumption of the Timing Verifier is that the circuit to be verified has periodic behavior. That is, given a circuit and a set of input stimulus, there is some state of the circuit S and some time T, such that starting the circuit in state S, applying the inputs and simulating for time T, the circuit returns to state S. (By state of a circuit, we mean the value history of each signal in the design.) Synchronous sequential circuits, and strictly combinational circuits both have this property.

THE IDEALIZED CLOCK CYCLE

The Timing Verifier tests for all timing errors over the course of a single clock cycle. This clock cycle is not any particular cycle, but an IDEALIZED cycle in which all possible transitions are analyzed and tested for timing errors. In a circuit that contains a 4-bit counter, for example, each of the four bits does not undergo a transition every clock cycle. But in any given clock cycle, the outputs of the counter that DO undergo transitions undergo them at the same time within that cycle.

The Timing Verifier doesn't care whether any particular bit is changing value at a particular moment in time. What the Timing Verifier cares about is that when ANY of the bits undergoes a transition, the timing behavior of that signal does not interfere with the timing specifications of the rest of the circuit.

To the Timing Verifier, therefore, all four bits change at the same time. That is to say that they all change during the same portion of the clock cycle. The signal history for each of the four bits of the counter output is identical in the Timing Verifier. Remember, this is not a functional model, but just a timing model.

When the Timing Verifier reports an error condition (let's say a setup time violation) at time 67.0 in a 100 ns clock cycle, it is not telling you that a setup time violation occurs during every clock cycle, but rather that during every clock cycle when this component IS active, a setup time violation occurs at the 67th nanosecond within that cycle.

If the particular component in question is only active once every several hundred cycles, then this error could be difficult to detect by other means.

1.3 SIGNAL VALUES IN THE TIMING VERIFIER

Because the SCALDsystem Timing Verifier does not require you to specify the exact value of each signal, but only that of the clocks, the Timing Verifier reports the value history of each signal using a system of eight signal values. The timing behavior of all synchronous circuits can be accurately represented using these eight signal values. Of these eight values, the three fundamental signal values are stable, changing, and unknown. The remaining five signal values are specific cases of the fundamental values.

Logic 1 and logic 0 are specific cases of a stable signal value, rising and falling are specific cases of a changing signal value, and high impedance is a specific case of an unknown signal value. The eight values are shown in Figure 1-2 with their abbreviations used when giving a signal's value history, and their waveform representation. These signal values are internal to the SCALDsystem Timing Verifier. On the basis of the clock behavior you specify and the timing models of each part in your design, the Timing Verifier calculates and reports the signal history for each signal.

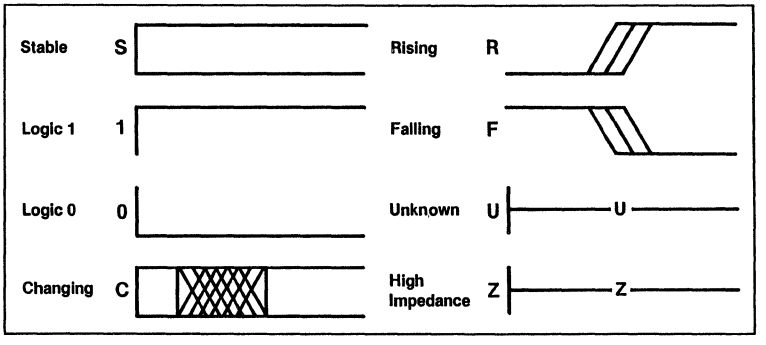

Figure 1-2: Signal Values

Notice that for timing verification these eight signal values are sufficient to describe all pertinent conditions. An "in-between" value such as 2.5V is not relevant to timing verification, nor are threshold voltages.

Here, as an example, is a signal DATA and its value history shown as a waveform representation and in list form:

DATA 0:0.0, R:15.0, 1:30.0, F:35.0, 0:45.0

Figure 1-3: Waveform and Signal History

This signal history means that the signal DATA might rise as soon as 15 ns, that it must reach the value 1 by 30 ns, and then remains high until 35 ns when it could fall and it must reach the value 0 at 45 ns. The signal remains low for the rest of the cycle. This signal history reports the behavior of the signal DATA during those clock cycles when DATA is active. This signal history does not mean that DATA is active during every clock cycle. The numbers given in the signal history do not therefore specify real time but relative time within the clock cycle. If for a clock period of 100 ns the signal DATA is active only every third period, then in real time DATA would be high from 230 ns to 235 ns and from 530 to 535 ns, and so on. But to the Timing Verifier this is not important. What is important to the Verifier is whether the signal DATA causes timing conflicts during the cycles when it IS active. That's why the Timing Verifier needs only to know about the cyclic behavior of DATA and not its behavior in real time.

For each part in the design, the Timing Verifier determines the signal value of the input signals and then uses a timing model to represent the requisite delays and to determine the signal value of the output. The timing models are produced by Valid engineers from a small set of Timing Verifier primitives. Each timing primitive uses a unique algorithm to determine the value of signal outputs. See Section 7, Timing Models for complete truth tables for each timing primitive.

Using the signal value stable (S) greatly reduces the amount of time spent preparing inputs for the Timing Verifier. A register, for example, changes state only during a short interval after it is clocked. The rest of the time it is stable. By specifying the signal names as having the value S (stable) the contents of registers and memories do not have to be individually specified. Using the signal value S also exponentially reduces the number of states that must be simulated to verify the timing behavior of the circuit. For example, a 16-bit counter that contains the value SSSSS....SS (16 times) has only one state, not 2**16 states. Rarely does a circuit's timing depend on the actual value in the counter, but merely how long after the counter is clocked it takes for the outputs to stop undergoing transitions.

For cases where correct modelling of the signals in a circuit requires specifying (0,1) behavior, the Timing Verifier uses a mechanism called **case analysis** explained later in this manual.

Signal values are calculated in picoseconds (1/1000 ns) and reported in nanoseconds with 0-3 decimal digits. To change the resolution of reported signal values use the OUTPUT_RESOLUTION directive.

SECTION 2
TIMING VERIFIER OPERATION

To perform timing verification, you must first define the periodic operation of the master clock and of any sub-clocks in the design. This includes specifying the clock period, duty cycle, and any clock skew that needs to be taken into account.

The Verifier then determines first at what time during the cycle each possible transition occurs, and then whether any of these transitions cause timing errors. The first part of this process is what we call "convergence" and the second is the actual error analysis. If the circuit does not converge, it is not synchronous and the Timing Verifier cannot effectively proceed.

2.1 RUNNING THE TIMING VERIFIER

Here is an overview of the steps required to run the Timing Verifier:

1. Specify the clock behavior with an assertion statement on the GED drawing or in the case.dat file. Be sure to **WRITE** the GED drawing.

2. Edit the Compiler directives file, the Verifier directives file and any other optional input files.

3. **verify**

4. Check tvlst.dat for Compiler and Timing Verifier errors. If there are process errors (Compiler, syntax, runtime), make the necessary changes to the drawing in GED, re-edit the necessary files, and **verify** again.

5. Look in tvlst.dat for signal history and information on design errors.

TIPS

The following tips are keyed to the steps in the list above.

1. It is important to specify the clock behavior before running the Timing Verifier. Without information about the master clock, the Verifier cannot produce meaningful results. For more information about Timing Assertions, see Section 4.

2. You need to edit the Verifier Directives file and specify the clock period and number of clock intervals. You should also enter the name of your drawing with the ROOT_DRAWING directive. The Verifier uses the ROOT_DRAWING directive to call the Compiler. If you don't enter your drawing name in the Verifier Directives file, you must enter it on the command line. See Section 3 for more information about Directives.

   Also, be sure you edit the Compiler Directives file and enter the directives you require before using the Timing Verifier for the first time on a design. See the Compiler Reference Manual for more information on the Compiler.

3. To run the Timing Verifier, you use the command:

   verify

   The name of the root drawing can be included as a command line argument, like this:

   verify subtractor

   When the root drawing name is included as a command line argument, the name entered there overrides any name given in the ROOT_DRAWING directive in the Verifier Directives file. When the root drawing name includes spaces, the name must

be enclosed in quotes on the command line, like this:

verify 'my subtractor'

4. Compiler errors are divided into three groups according to severity: errors, warnings, oversights. Errors are the most severe and must be corrected before proceeding with timing verification.

   Timing Verifier errors are divided into three groups: syntax errors, runtime errors, and timing errors. Syntax and runtime errors must be corrected before consulting verification results. Timing errors are errors the Verifier found in the design. Syntax and runtime errors often cause spurious timing errors. For detailed error information, see Section 8.

5. The Timing Verifier is an analysis tool that is used not just to discover timing errors, but also to produce detailed timing information about the design. This detailed timing information is called signal history and appears in the listing file (tvlst.dat). For more information on signal history, see section 2.4 below.

   To use the Timing Verifier effectively, check carefully for Compiler, syntax, and runtime errors before consulting the signal history or running Plottime. The signal history reported in the listing file is usually inaccurate when these types of errors are reported.

   Results of the verification are presented in two forms. The first is a list of each signal by name giving the signal's values over a clock cycle (we call this the signal's "history"). The second is a file which when used as input to the Plottime program produces waveform representations of each signal over a clock cycle. The waveform output is particularly useful for spotting timing conflicts. For more information on Plottime, see the Plottime Reference Manual.

The listing file also contains information about design errors the Verifier found. These design errors are called "timing errors" or timing violations. For more information about timing violations, see below, and Section 8.

2.2 THE VERIFICATION PROCESS

The Timing Verifier operates in two phases. It first determines at what time during the cycle each possible transition occurs, and then whether any of these transitions cause timing errors. The first part of this process is what we call "convergence" and the second is the actual error analysis. If the circuit does not converge, it is not synchronous and the Timing Verifier cannot effectively proceed.

CONVERGENCE

The Timing Verifier makes multiple evaluation passes until the circuit converges to a steady behavior. During the first evaluation pass, all time primitives are evaluated. After that, the Timing Verifier checks the inputs of the time primitives for change. If any input changes, another evaluation pass is made. In this pass only the primitives that had any inputs change are evaluated. The cycle continues until all of the time primitive inputs are unchanged. When this occurs, the circuit has "converged" to a steady behavior. This is an "event-driven" algorithm.

Circuits with much feedback require more evaluation passes because the time primitive inputs fluctuate. The critical factor is the feedback in the circuit, not the size of the circuit. Timing violations cannot be determined if the circuit does not converge because stable circuit behavior has not been determined. If the circuit does not converge an error occurs when the maximum number of evaluation passes (default is 2000) is reached.

TIMING VIOLATIONS

After the circuit converges, the Timing Verifier checks for timing violations. There are time primitives that correspond to each timing violation. The "setup hold" primitive, the "setup rise hold fall" primitive, the "edge to edge" primitive, and the "min pulse width" primitive check for setup and hold time, pulse separation, and minimum pulse width timing constraints respectively. The Timing Verifier also checks for any signal delays that are greater than the clock period. For more information on timing errors see Section 8, Error Documentation, and Section 7, Primitives under the specific primitives.

2.3 TIMING VERIFIER FILES

This section describes the Timing Verifier Input and Output files.

INPUT FILES

The Timing Verifier has one required and three optional input files. The required input file is the Verifier Directives file. The optional input files are:

Case file
Wire Delay file
Compiler Expansion file (produced by the 7.27 or earlier Compiler in a compilation for time).

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

Verifier Directives File	=	**verifier.cmd**
Case Analysis File	=	**case.dat**
Wire Delay File	=	**delay.dat**
Compiler Expansion File	=	**cmpexp.dat**

Here is a description of each input file. For file names under VMS and CMS, see File Names later in this section.

Verifier Directives File

The Verifier Directives file (verifier.cmd) is the only required input file to the Timing Verifier. Directives are used to give commands to the Timing Verifier covering a broad range of topics. Here is a sample Verifier directives file:

```
CLOCK_PERIOD 132.0;
CLOCK_INTERVALS 12;
CLOCK_SKEW 1.0;
PREC_CLOCK_SKEW 0.1;
WIRE_DELAY 0.0-2.0;
PRINT_WIDTH 80;
RECONV_FANOUT ON;
ROOT_DRAWING TEST;
END.
```

Note that all Timing Verifier directives must be separated by a semi-colon (;) and last line of the file reads

```
END.
```

Don't forget the final period. Upper case or lower case may be used for the directives. The Timing Verifier does not pay any attention to the end-of-line character or to multiple spaces. Comments may be placed in the file enclosed in curly braces { }. Nested comments are not permitted.

There are a great many Timing Verifier directives most of which take default values. Each directive is described in Section 3, Directives.

Case File

The Case file **case.dat** is used for two main purposes:

1. To specify timing assertions for signals in a design when it is inconvenient to enter them on the drawing.

2. To specify several different cases of signal values to be evaluated. Each case is evaluated in turn.

For a more complete description of case.dat, see under Case Analysis in Section 4, Timing Assertions.

Wire Delay File

The Wire Delay file **delay.dat** is used to specify individual wire delays on a net by net basis. Most often, this file is used to feed back wire delay data from the physical design system to the Timing Verifier. When the Wire Delay file is used in this function, other mechanisms for specifying delay should be turned off. For more information, see Section 5, Delays.

Compiler Expansion File

When the ROOT_DRAWING directive is omitted, the Timing Verifier does not call the Compiler/Linker. Instead, it looks for a Compiler Expansion file (cmpexp.dat) to provide the design input. When the ROOT_DRAWING directive is included, any existing Compiler Expansion file is ignored. See the Compiler Reference Manual for additional details.

OUTPUT FILES

The Timing Verifier produces three output files:

Listing File
Log File
Waveform Input File

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

Verifier Listing File	=	**tvlst.dat**
Verifier Log File	=	**tvlog.dat**
Waveform Input File	=	**plotsig.dat**

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

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

Listing File

The Listing file **tvlst.dat** is the most important of the three Verifier output files. It contains process information on the Verifier run, error information, and the Verification results in the form of signal history and, if requested, histograms. It is very important to check the error information in this file before using the verification results, as some Verifier errors result in invalid output from the Timing Verifier. The Verifier reports three types of errors: syntax errors, run time errors, and timing errors. Syntax and runtime errors result in invalid Verifier output. Signal history is explained in detail in the next section. Complete error message documentation appears in Section 7.

Log File

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

Waveform Input File

The Waveform Input File **plotsig.dat** is an ASCII file that serves as the input file to the program Plottime which produces waveform diagrams of the signals from the design verified. Plottime produces waveform diagrams from a Simulator output file as well as from a Timing Verifier output file. For more information on Plottime, see the Plottime Reference Manual.

FILE NAMES

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

Table 2-1. Timing Verifier Input and Output Files

File Name	UNIX	VMS	CMS
Directives	verifier.cmd	VERIFIER.CMD	VERIFIER CMD
Case	case.dat	CASE.DAT	CASE DATA
Wire Delay	delay.dat	DELAY.DAT	DELAY DATA
Expansion	cmpexp.dat	CMPEXP.DAT	CMPEXP DATA
Listing	tvlst.dat	TVLST.DAT	TVLST DATA
Log	tvlog.dat	TVLOG.DAT	TVLOG DATA
Waveform	plotsig.dat	PLOTSIG.DAT	PLOTSIG DATA

2.4 SIGNAL HISTORY

The Verifier listing file, tvlst.dat, includes a section called the signal history. This section lists each of the named signals in the design and additional other groups of signals as requested. The LIST directive regulates the order in which the named signals appear and whether or not other groups of signals are included.

The LIST directive is an optional directive that is entered in the Verifier Directives File. For more information see Section 3, Directives.

The signal history of a signal is a list of the values of that signal (0, 1, S, R, F, C, U and Z) and the times (over a single clock cycle) when the signal has each value. An example of signal history is:

DATA S:0.0 R:18.2 1:20.0 F:31.0 0:32.8

If the clock period is 80ns, the signal DATA is STABLE at time 0.0, then starts to RISE at 18.2 ns, stabilizes at 1 (high) at 20.0 ns, starts to FALL at 31.0 ns, then is 0 (low) again starting at 32.8 ns.

Signal history always describes cyclical behavior. This is the behavior of the signal DATA during every clock period in which it is active.

Figure 2-1 shows the signal history portion of a Timing Verifier listing file. On the left are the signal names, and on the right are the different values of each signal over the course of a complete clock period. The initial value (at time 0.0) is given for each signal, and additional values are given for all signals that change value during the cycle.

The signals are listed in alphabetical order. A signal named "Z" would appear at the end of this list, after the unnamed signals.

Signals that are tied to 0 and 1 on a design have very uninteresting signal histories. For reporting purposes in the signal history, these signals are bundled together into two groups and renamed, respectively, to TV_0 and TV_1.

Notice the signal name A <3 . . 0> at the top of the left column. This is a bus signal that is four-bits wide. The Timing Verifier consolidates signal names that are bits of a bus and have identical signal history. The design may have a signal with this name, or it may, instead, have four signals with the names A <3>, A <2>, A <1>, and A <0>. Whichever notation is used, when the bits of the bus have identical signal history, they are grouped together and reported in the signal history as a bus signal, such as A<3 . . 0>. This keeps the signal history from getting too long.

The first column on the right gives the initial signal values for all of the signals. Each signal for which no timing behavior was specified was set to STABLE (S) at time 0.0. In this case, all signals except the CLOCK and the signals tied to 1 and 0 (represented by TV_1 and TV_0) are set to STABLE.

```
Values of all signals

A<3..Ø> . . . . . .  S:Ø.Ø
B<3..Ø> . . . . . .  S:Ø.Ø
CLOCK !C Ø-4 . . . . .  1:Ø.Ø, Ø:28.Ø
SUM<3..Ø> . . . . . .  S:Ø.Ø, C:16.Ø, S:51.Ø
TV_Ø . . . . . .  Ø:Ø.Ø
TV_1 . . . . . .  1:Ø.Ø
UN$1$ADDR$5P$A . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$ADDR$5P$A$1 . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$ADDR$5P$A$2 . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$ADDR$5P$A$3 . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$ADDR$5P$B . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$ADDR$5P$B$5 . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$ADDR$5P$B$6 . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$ADDR$5P$B$7 . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$ADDR$6P$A . . . . .  S:Ø.Ø, C:13.5, S:42.Ø
UN$1$ADDR$6P$A$1 . . . . .  S:Ø.Ø, C:13.5, S:42.Ø
UN$1$ADDR$6P$A$2 . . . . .  S:Ø.Ø, C:13.5, S:42.Ø
UN$1$ADDR$6P$A$3 . . . . .  S:Ø.Ø, C:13.5, S:42.Ø
UN$1$ADDR$6P$Y . . . . .  S:Ø.Ø, C:21.Ø, S:38.Ø
UN$1$ADDR$6P$Y$11 . . . . .  S:Ø.Ø, C:21.Ø, S:38.Ø
UN$1$ADDR$6P$Y$12 . . . . .  S:Ø.Ø, C:21.Ø, S:38.Ø
UN$1$ADDR$6P$Y$13 . . . . .  S:Ø.Ø, C:21.Ø, S:38.Ø
UN$1$DFF$7P$D . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$DFF$7P$D$1Ø . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$DFF$7P$D$11 . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$DFF$7P$D$9 . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
UN$1$DFF$8P$Q . . . . .  S:Ø.Ø, C:8.5, S:27.Ø
```

Figure 2-1. Signal History

When the signal history shows signals whose values remain stable for an entire clock cycle, it can mean that the circuitry is not being fully tested and that the Verifier needs some additional information.

LOW-ASSERTED SIGNALS IN THE SIGNAL HISTORY

Low-asserted signals do not always appear in the signal history in exactly the same form in which they appear on the GED drawing.

To indicate a low-asserted signal on a drawing using the SCALD Language (format 1) you use an asterisk * following the signal name, like this:

CLOCK*

The internal data base of the SCALDsystem, however, represents the low-asserted signal name CLOCK* as -CLOCK in its data base. Since it is this data base that gets passed on to the Timing Verifier (and the Packager and the Logic Simulator), the names you see in the signal history do not exactly match the names on your GED drawing. A signal named

NAME*

appears as

- NAME

in the signal history.

For more details on signal name syntax and formats, see the SCALD Language Reference Manual.

2.5 USER SPECIFIED TIMING INFORMATION

Digital systems are composed of components and their interconnections. Complete timing verification includes the verification of all component timing constraints, and the verification of all interconnect timing constraints. The Timing Verifier obtains component timing information from the timing models provided in Valid libraries for each component. The interconnect timing information must be specified by the user. There are two basic types of interconnect timing information:

1. Periodic information: This is the specification of the periodic behavior of clock and interface signals.

2. Wire Delay: This includes all delays resulting from the ways components are interconnected.

Periodic information is specified to the Timing Verifier in the form of clock and signal assertions. The periodic behavior of the clock signals in a design must be specified for the Timing Verifier to be able to produce meaningful results. The periodic behavior of interface signals and primary input and output signals provides additional useful information to the Verifier, but is not required. The sections on Assertions and Case Analysis later in this manual describe the ways to specify periodic information.

Wire delay information is not required input to the Timing Verifier, but including it can greatly increase the accuracy of the timing information obtained. In early design phases when detailed wire delay information is not available, estimated wire delays can be included or the Delay Estimator can be used to estimate load dependent delays. After the design has been sent to a physical design system, delay information from that system can be fed back as input to the Timing Verifier. The section on Delays later in this manual describes the ways to specify interconnect delay information. The directives that regulate delay usage and the Delay Estimator are described in the Directives section.

WAYS TO SPECIFY PERIODIC AND DELAY INFORMATION

There are three different ways to provide user specified information to the Timing Verifier. These are:

1. Enter the data on the GED drawing.

2. Specify a value for a directive in verifier.cmd.

3. Enter the data in an input file (case.dat or delay.dat).

Frequently you may choose which method to use. Clock assertions, for example, can be added on the GED drawing, or they can be entered in the case.dat file. In both cases, the clock period and number of clock intervals must be entered in the directives file.

Wire delays can also be added on the GED drawing, or a global default delay can be set by using a directive. In addition, the delay estimator can be used to calculate load dependent delays. This feature is regulated by a directive in the verifier.cmd file.

The multiplicity of choices can be confusing at first, but the options are provided to give you complete control of the assertion information and interconnect delay information used by the Verifier. For example, early in the design process you can specify wire delay on the GED drawing to provide an early approximation. Later, when more accurate data is available, you can then disable these approximations without changing the drawing. Generally speaking you want to add timing information that you will not be changing frequently directly on the GED drawing. Other information is more easily added in an input file.

The next three sections of this manual cover first the Timing Verifier directives, then assertions, and delays.

SECTION 3
TIMING VERIFIER DIRECTIVES

The Timing Verifier accepts a large number of directives that regulate many aspects of the verification process. These directives fall into five basic categories: general directives, execution and output directives, delay directives, delay estimator directives, and technology-linked directives. The first two categories contain directives that you will use on a regular basis. The directives in the third group are used to regulate the calculation of wire delays. A great number of options are available by combining these directives. The directives in the fourth group are used only when using the Delay Estimator. The directives in the fifth group are used to select alternative timing functions for certain Timing Verifier primitives and to conform to the requirements of specific technologies.

Each directive is described below. They are listed alphabetically under the appropriate group.

3.1 GENERAL DIRECTIVES

These directives cover clock behavior and other general Timing Verifier functions. They are the most commonly used Verifier directives.

This category includes the following directives:

CLOCK_PERIOD
CLOCK_INTERVALS
CLOCK_SKEW
PREC_CLOCK_SKEW
RECONV_FANOUT
ROOT_DRAWING
TIMING_DIAGRAMS

CLOCK_PERIOD

Sets the period of the clock used by the Timing Verifier. This clock period is used in timing assertions. Clock period is specified in units of nanoseconds. The directive

CLOCK_PERIOD 56.0;

sets the clock period to 56 ns. If unspecified, the Timing Verifier sets the period to 100 ns.

CLOCK_INTERVALS

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

CLOCK_PERIOD 100.0;

sets the clock period to 100 ns, and the directive

CLOCK_INTERVALS 20;

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

For example, the signal MASTER CLK !C 0-10,15-20 is high for the first ten intervals, (that is, from 0 ns to 50ns) and then high again for the last five intervals, from 75.0 ns to 100 ns. The signal history for this clock is:

1:0.0, 0:50.0, 1:75.0

If unspecified the clock is divided into eight intervals.

CLOCK_SKEW

Specifies the amount of symmetrical clock skew to be added to signals having an !C timing assertion. CLOCK_SKEW is skew from the nominal time (in nanoseconds). The directive

CLOCK_PERIOD 100.0;

sets the clock period to 100 ns, the directive

CLOCK_INTERVALS 20;

divides the clock into 20 intervals of equal length, and the directive

CLOCK_SKEW 0.1;

sets the skew to -0.1, and + 0.1 ns from the nominal.

With clock skew added, the signal history for the signal MASTER CLK !C 0-10,15-20 becomes:

1:0.0, F:49.9, 0:50.1, R:74.9, 1:75.1

If unspecified the CLOCK_SKEW is 0 ns.

PREC_CLOCK_SKEW

The directive PREC_CLOCK_SKEW is identical to CLOCK_SKEW except it affects only signals with !P timing assertions.

If unspecified the PREC_CLOCK_SKEW is 0 ns.

RECONV_FANOUT

This directive tells the Timing Verifier to analyze the circuit to understand reconvergent fanout, The use of this directive eliminates spurious errors that are currently flagged for circuits that count on correlated signal skews to work. The default for this directive is ON.

Reconvergent fanout is where a signal fans out from a common point in a circuit through different paths, which then reconverge at some other point. When these signals come together at a primitive like a setup-hold checker, the skew which is common to them needs to be subtracted out before the check is done, or else spurious errors may be generated.

One of the most common cases of this is a shift register. Skew on the clock to the register is common to all of the bits of the register and needs to be subtracted out before checking the setup and hold times of the shift bits.

Other common terms for reconvergent fanout are correlated skews and common ambiguity. The directive

RECONV_FANOUT ON;

causes the Timing Verifier to take into account correlated signal skews. If unspecified, the reconvergent fanout analysis is performed.

ROOT_DRAWING

This directive is used to specify the name of the drawing that you want to run timing verification on. Use quotes to enclose the drawing name, like this:

ROOT_DRAWING 'subtractor';

When a drawing name is specified on the command line it overrides the name given with this directive.

This directive is new for the ValidTIME 1.0 Timing Verifier. If this directive is omitted, the Linker/Compiler is not called. Instead, the Verifier looks for a Compiler expansion file (cmpexp.dat) from which to read the design.

TIMING_DIAGRAMS

When set to ON, this directive instructs the Timing Verifier to produce the output file plotsig.dat. This output file is used as input to the Plottime program to produce Timing Diagrams. The directive

TIMING_DIAGRAMS ON;

generates waveform input file, plotsig.dat If unspecified, this directive is OFF.

3.2 EXECUTION AND OUTPUT DIRECTIVES

These directives regulate the number and types of errors allowed during execution and the format and content of listing files. This category includes the following directives:

DIFF_PASSES
LIST
MAX_ERRORS
MAX_EVAL_PASSES
MAX_EXP_ERRORS
OUTPUT_RESOLUTION
PRINT_WIDTH

DIFF_PASSES

This directive aids debugging when a circuit loops indefinitely instead of converging to a stable behavior. The directive takes an integer as its value, as in:

DIFF_PASSES 3;

The default value for this directive is zero.

When the DIFF_PASSES directive has a value other than zero, the Timing Verifier continues for the specified number of passes after the number of passes specified with the MAX_EVAL_PASSES directive. For these additional passes, the Verifier prints the signals that are changing value. This information is very helpful in locating zero-delay feedback loops that cause the problem.

When MAX_EVAL_PASSES is 500, and DIFF_PASSES is 3, the following output can appear:

Signals that have changed after pass 500

DATA 1:0.0

Signals that have changed after pass 501

DATA 0:0.0

Signals that have changed after pass 502

DATA 1:0.0

The user is alerted to check the signal DATA in the design to see why it is oscillating.

LIST

This directive takes one or several options separated by commas. These options control the way in which the signal history is presented in the listing file. These options are:

1. BY_NAME/NOBY_NAME (default BY_NAME) BY_NAME causes the signals to be listed sorted alphabetically by name. NOBY_NAME causes signals to be sorted by path name. Signals with unique names are listed by name only, and signals with multiple path names are listed with each path name indented:

   ```
   CLRINIT !C 0+ 1.0
     (TST123221 .123.10P)        1:0.0, 0:1.0
     (TST123221 .221.9P)         1:0.0, 0:1.0
   ```

   This eliminates many superfluous path names from the signal listing.

2. CHIP/NOCHIP (default NOCHIP) CHIP causes local signals within timing models to be listed.

3. CONSTANT/NOCONSTANT (default NOCONSTANT) CONSTANT causes constant signals (TV_0, TV_1) to appear in the listing file (tvlst.dat) and in the Plottime input file (plotsig.dat). Constant signals are of little interest in the signal history or in

timing diagrams because they do not change value.

4. DOT/NODOT (default NODOT) DOT causes signals that are generated automatically for the inputs of dot gates (wire gates) to be listed.

5. HISTOGRAM/NOHISTOGRAM (default NOHISTOGRAM) HISTOGRAM causes bar charts showing timing error statistics to be generated.

6. NC/NONC (default NONC) NC causes NC or "not connected" signals to be listed.

7. RISE_FALL/NORISE_FALL (default NORISE_FALL) RISE_FALL causes both the rise and the fall skew associated with each time/value pair to appear in the signal history in parentheses. The first number in the parentheses is the rising skew, and the second is the falling skew. Here is an example:

   DATA S:0.0, C:10.0(1.0,0.0), S:20.0

   In this example the signal DATA has the value CHANGING between time 10 and time 20. In addition, if the transition is rising (from 0 to 1), it may occur as late as time 21 (20 + 1.0 ns rise skew).

   When the default value (NORISE_FALL) is used for the same example, the signal history reads:

   DATA S:0.0, C:10.0, S:21.0

8. SKEWS/NOSKEWS (default NOSKEWS) SKEWS causes the common skew associated with each signal to appear at the beginning of the signal history for that signal, and does not include the effect of that skew in the signal history. The signal history will

generally have fewer entries in it. For example, for a signal DELAY with a skew of + 2.70 and the directive LIST SKEWS, here is the signal history:

DELAY (+ 2.70) 1:0.0, 0:8.95, 1:13.90, 0:28.90, 1:33.90

With the default value NOSKEWS, the signal history is:

DELAY 1:0.0, F:8.95, 0:11.65, R:13.90, 1:16.60, F:28.90, 0:31.60, R:33.90, 1:36.60

9. STRENGTH/NOSTRENGTH (default NOSTRENGTH) STRENGTH causes the signal strength (hard, soft, undriven) to appear in square brackets after each signal value and before the time in the signal history. The three possible signal strengths are designated as h (hard), s (soft), or z (undriven). Here is an example:

   DATA 0[h]:0.0, 1[h]:10.0, 0[h]:30.0

10. TRAN_INPUT/NOTRAN_INPUT (default NOTRAN_INPUT) Transmission gates have two bi-directional pins, T1 and T2. TRAN_INPUT causes the Verifier to list the values that bi-directional transmission gates see at their pins T1 and T2 that the other drivers on those nets have generated. This feature is useful in debugging complicated circuits with a lot of bi-directional transistors in them.

11. UNNAMED/NOUNNAMED (default NOUNNAMED) UNNAMED causes unnamed signals to be listed in the output listing.

12. VIOLATIONS/NOVIOLATIONS (default NOVIOLATIONS) VIOLATIONS causes a report of all types of timing violations to be printed. The violations are sorted in order of descending severity. Here is an example:

Table 3-1. Setup Time Violations

Violation	Time	Error #	Primitive
52.0	509.5	11	(TIM11P SUH4P)
30.1	546.0	22	(TIM11P SUH14P)
17.5	546.0	20	(TIM11P SUH4P)
7.0	583.0	12	(TIM11P SUH4P)
2.7	6.9	15	(TIM11P SUH14P)
2.7	6.9	9	(TIM11P SUH4P)
2.6	595.0	17	(TIM11P SUH4P)

Included in this report are: setup and hold violations, low and high pulse width violations, and minimum and maximum edge-to-edge timing violations.

An example LIST directive is:

```
LIST UNNAMED, NODOT, VIOLATIONS,
  HISTOGRAM, NOBY_NAME, CHIP;
```

MAX_ERRORS

If more than MAX_ERRORS occur during the reading of the input files to the Timing Verifier, verification is aborted. The input files to the Timing Verifier are the Compiler expansion file (cmpexp.dat), case.dat, and delay.dat. If there are a large number of errors in the input files, the Timing Verifier will not produce meaningful results. This directive saves time by aborting the program before verification of the entire design and reporting the errors found in the input files. The correct syntax for this directive is:

```
MAX_ERRORS 2;
```

If unspecified, MAX_ERRORS defaults to 10. The MAX_ERRORS directive is not triggered by timing errors found in the design. It is only triggered by errors found in the input files to the Timing Verifier.

MAX_EVAL_PASSES

If more than MAX_EVAL_PASSES occur during the verification of the design then verification is aborted. The directive

```
MAX_EVAL_PASSES 50;
```

aborts current Verifier run if there are more than fifty passes. A runtime error is reported. If unspecified, MAX_EVAL_PASSES is 2000. If a non-integer value is given to this directive, the Verifier rounds it to the closest integer.

MAX_EXP_ERRORS

If more than MAX_EXP_ERRORS are detected in the Compiler expansion file then verification is aborted. The Compiler expansion file is generated by the Compiler and can only have errors in it if edited by a user. The directive

```
MAX_EXP_ERRORS 4;
```

aborts this run if there are more than four errors in the expansion file. If unspecified, MAX_EXP_ERRORS is 0.

OUTPUT_RESOLUTION

All Timing Verifier calculations are made in picoseconds. The output data is given in nanoseconds to one, two, or three decimal places. The OUTPUT_RESOLUTION directive specifies the number of decimal places used in output data. There are four possible values for the OUTPUT_RESOLUTION directive:

```
OUTPUT_RESOLUTION 0;
OUTPUT_RESOLUTION 1;
OUTPUT_RESOLUTION 2;
OUTPUT_RESOLUTION 3;
```

When the first value is used, this directive specifies that output data includes no decimal places (0). The second value specifies one decimal place (0.0). The third value

specifies two decimal places (0.00). The fourth value specifies three decimal places (0.000).

PRINT_WIDTH

This directive tells the Timing Verifier how many columns may be used in the listing file. The output is formatted according to this specification. The directive

```
PRINT_WIDTH 80;
```

formats output for an 80 column display. If unspecified, the width is 132. (Only 80 and 132 are permitted.)

3.3 DELAY DIRECTIVES

This category includes the following directives:

```
DELAY_MODEL
RISE_FALL_ANAL
RISE_FALL_MODELS
USE_DRAWING_WD
WIRE_DELAY
```

DELAY_MODEL

The DELAY_MODEL directive takes one of three possible values:

```
MIN/MAX
MIN
MAX
```

MIN/MAX is the default and tells the Timing Verifier to use both the minimum and maximum available delays.

MAX tells the Timing Verifier to use only maximum delays.

MIN tells the Timing Verifier to use only minimum delays.

The DELAY_MODEL directive works with the RISE_FALL_MODELS directive to let you use six combinations of delay data. When the DELAY_MODEL directive is MIN/MAX (the default) and the RISE_FALL_MODELS directive is ON (the default), all of the available delay data is used.

Take as an example a signal that has 1 to 3 ns delay rising and 2 to 4 ns delay falling. If we attach that wire delay to the signal name we record it in this form:

\WD (1-3, 2-4)

The chart below shows what delay the Timing Verifier selects.

Table 3-2. Delays Selected

R-F	DELAY MODEL	DELAY USED	COMMENTS
ON	MIN-MAX	(1-3, 2-4)	all
ON	MAX	(3,4)	max rise, max fall
ON	MIN	(1,2)	min rise, min fall
OFF	MIN-MAX	(1-4)	min of min, max of max
OFF	MAX	(4)	highest max
OFF	MIN	(1)	lowest min

RISE_FALL_ANAL

This directive toggles on and off. When on, the directive tells the Timing Verifier to perform rise/fall delay analysis through cascades of inverting logic not only on signals whose 0/1 behavior is known, but also on signals whose behavior is reported as stable/changing. The Verifier does this by selecting the worst case delay for the first component, remembering whether that delay was a rise delay or a fall delay, and selecting the opposite type of delay the next time it encounters inverting logic on that signal path.

Rise/fall delay analysis is always performed for signals whose 0/1 behavior is known when they encounter asymmetrical rise/fall delays. When no asymmetric rise/fall

delays exist, or when the asymmetric delays have been made symmetric by turning off the RISE_FALL_MODELS directive, no rise/fall analysis is performed on any of the signals.

When RISE_FALL_MODELS is OFF, RISE_FALL_ANAL is set to OFF by the Timing Verifier.

When RISE_FALL_ANAL is OFF and RISE_FALL_MODELS is ON, rise/fall analysis is performed on signals whose 0/1 behavior is known, but not on signals whose behavior is only known to be changing/stable. The directive

RISE_FALL_ANAL ON;

causes the timing verifier to exploit different rise and fall delays for stable/changing values of signals. If unspecified, the directive is ON.

RISE_FALL_MODELS

This directive toggles on and off. The directive tells the Timing Verifier to select both rise and fall delays from the delay data provided. The way in which the rise and fall delay values are selected is regulated by the DELAY_MODEL directive. If this directive is OFF, rise and fall delays are symmetric. If this directive is turned off, then RISE_FALL_ANAL is also turned off. The directive

RISE_FALL_MODELS ON;

selects both rise and fall delays from the delay data. If unspecified, the directive is ON. See above under DELAY_MODEL for the interaction of the DELAY_MODEL directive and the RISE_FALL_MODELS directive.

USE_DRAWING_WD

This directive regulates whether the Timing Verifier uses wire delay data entered on the drawing either in signal names or with the WIRE_DELAY property. The directive toggles on and off. When this directive is on, wire delay on the drawing overrides the wire delay set in the WIRE_DELAY directive. By default this directive is on.

When using the delay estimator, this directive should usually be turned off, to prevent delays on the drawing from being added to the delays calculated by the delay estimator.

This directive does not regulate the use of delay entered on the drawing with the CHIP_DELAY and CLOCK_DELAY properties. When, for example, CHIP_DELAY is used to add 4 ns of delay to a pin of a component, and the USE_DRAWING_WD directive is OFF, the 4 ns of delay is still used by the Timing Verifier in delay calculations. This is consistent with the function of CHIP_DELAY to specify delay in timing models and with the function of CLOCK_DELAY to describe tuned clocks. The directive

```
USE_DRAWING_WD OFF;
```

turns off use of wire delays specified on the drawing. If unspecified, this directive is on.

WIRE_DELAY

This directive assigns a default wire delay to each net in the design that is connected to at least one input pin of a device and for which wire delay is not specified by a property (either attached with the property command or added to the signal name). By default the WIRE_DELAY directive takes the value 0.0-0.0. When omitted, the directive takes the default value.

The value of the WIRE_DELAY directive is specified in nanoseconds. When the directive takes a single numeric value (as 2.35), that value is added to each applicable net. When the directive takes a range of values (as 2.35-4.312), a range of delay is assigned to each net.

If a range is specified where the minimum delay is larger than the maximum delay, a syntax error occurs and the Verifier uses the larger number as a single numeric delay value. The directive

```
WIRE_DELAY 0.1-2.0;
```

assigns wire delay of 0.1 ns (min) to 2 ns (max) to all nets where the wire delay is not specified by a property.

3.4 DELAY ESTIMATOR DIRECTIVES

This category includes the following directives:

```
DEFAULT_DRIVE
DELAY_ESTIMATOR
LOAD_COEFFS
WIRE_ESTIMATE
```

DEFAULT_DRIVE

This directive defines the default drive to be used by the delay estimator when no DRIVE body property is given for a primitive. The value of this directive can be a single number, a range of numbers, or two ranges of numbers. When two ranges of numbers are given the first specifies min/max rise delay and the second specifies min/max fall delay. Here is an example:

```
DEFAULT_DRIVE 0.5-1.2, 0.4-1.0;
```

The default value for this directive is 0. When the directive is missing, the default value is used.

DELAY_ESTIMATOR

This directive is used to turn the delay estimator on or off. The directive

```
DELAY_ESTIMATOR ON;
```

turns the delay estimator on. If this directive is omitted, the delay estimator is set to OFF.

LOAD_COEFFS

This directive is used to specify the drive factor and coefficients used by the Delay Estimator to calculate delays when the load exceeds the drive factor.

For example:

When L = Total Load
D = Drive
F = Drive Factor

and delays need to be calculated according to these specifications:

1. L <= F
   Load Delay = D * L

2. F < L <= 2F
   Load Delay = D * F + 1.5 * D * (L-F)

3. 2F < L <= 3F
   Load Delay = D * F + 1.5 * D * F + 3 * D * (L-2F)

4. 3F < L <= 4F
   Load Delay = D * F + 1.5 * D * F + 3 * D * F + 4 * D * (L-3F)

Specify the LOAD_COEFFS directive as follows:

```
LOAD_COEFFS F, 1.5, 3, 4;
```

F is the value of the drive factor. Thus, the first number is the drive factor, the second number is the coefficient used in the second case, the third number is the coefficient used in the third case, and so on. Up to 99 coefficients may be specified.

The directive also takes an optional family specification. The family specification works the same as for the WIRE_ESTIMATE directive, see below for details.

When the LOAD_COEFFS directive is not specified, the normal delay estimator calculation is made, that is:

Load Delay = Drive * Total Load

WIRE_ESTIMATE

To estimate wire delay, the number of stops on each net is counted. The number of stops is converted to equivalent loads by using a lookup table specified by this directive. The WIRE_ESTIMATE directive takes for values a list of fixed point numbers and an optional FAMILY specification. A net with j stops receives a wire delay estimate given by the jth number in the list. Each wire estimate list may have a maximum of 100 entries. For nets with more than 100 stops, the last entry is used for the remining stops.

The family specification allows for a number of different WIRE_ESTIMATE tables to be used in the same Timing Verification run. When the FAMILY property is attached to a primitive, the WIRE_ESTIMATE table with the same FAMILY specification is used. When no FAMILY property is attached to a primitive, then the WIRE_ESTIMATE table without a FAMILY specification is used. Family is an identifier up to 16 characters long. Spaces are not allowed. An example set of WIRE_ESTIMATE directives are given below:

```
WIRE_ESTIMATE 1.0, 2.0, 3.0, 4.0;

WIRE_ESTIMATE ECL: 0.5, 1.0, 2.0, 3.0;

WIRE_ESTIMATE TTL: 1.0, 2.0, 3.1, 4.0;

WIRE_ESTIMATE ON_GATE_ARRAY: 0.3, 0.6, 1.0, 1.3;

WIRE_ESTIMATE BET_GATE_ARRAY: 1.0, 2.0, 3.1, 4.5;
```

If unspecified, the default is 0.0;

3.5 TECHNOLOGY-LINKED DIRECTIVES

This category includes the following directives:

DOT_TYPE
LATCH_ERR_MODEL
NC_SIGNALS
PULSE_FILTER
SET_MIN_DELAYS
TIMING_SIM_MODE
TS_BUS_TYPE

DOT_TYPE

Certain physical devices can be connected together to form wire-gates. The timing models for these physical devices include the pin property OUTPUT_TYPE on dottable outputs. The OUTPUT_TYPE property informs the Verifier what type of wire gate to form (OE, OC, or TS for open emitter, open collector, and tri-state), and what logic function is performed when those outputs are connected together (AND, OR, TS).

When the OUTPUT_TYPE property is missing, or when outputs with dissimilar logic function specifications are connected together, the Verifier needs to make some choice for the logic function of the bus. This choice is specified with the DOT_TYPE directive. The possible values for this directive are: DOT_AND, DOT_OR, and DOT_TS. The default value of this directive is DOT_AND. The directive

DOT_TYPE DOT_OR;

sets unspecified wire gates to type DOT-OR, the directive

DOT_TYPE DOT_AND;

sets unspecified wire gates to type DOT-AND, and the directive

DOT_TYPE DOT_TS;

sets unspecified wire gates to type tri-state.

If this directive is missing, unspecified buses are set to DOT_AND.

When the DOT_TYPE DOT_TS; directive is specified, the Verifier uses the tri-state model specified by the TS_BUS_TYPE directive.

LATCH_ERR_MODEL

This directive changes the model used for latches. There are three models for latches, OPEN, CLOSED, and CONSERVATIVE. The differences between these models is defined in the definition of the latch model in the section on Time Primitives. The default value for this directive is CONSERVATIVE. The directive

LATCH_ERR_MODEL CLOSED;

uses the closed model for all latches in the design. If this directive is omitted, the value defaults to CONSERVATIVE.

NC_SIGNALS

This directive regulates what value NC (non-connected) inputs are set to. There are 5 possible values, 0, 1, S, ASSERTED, and DEASSERTED. The values 0 and 1 set all non-connected inputs to either 0 or 1, ignoring any bubbled inputs. For example, 0 makes a bubbled input true, and a non-bubbled input false. A value of S sets all non-connected inputs to stable. The value ASSERTED sets all inputs to true (i.e., bubbled inputs to zero, and non-bubbled inputs to one), and DEASSERTED sets all inputs to false. For ECL circuits, DEASSERTED is generally the recommended value for this directive because of the resistor pull-downs on the inputs. The directive

NC_SIGNALS DEASSERTED;

sets non-connected inputs to deasserted. When unspecified, the default value is S.

PULSE_FILTER

This directive turns a pulse filter on that is used by the register and latch models. If a pulse has a large amount of skew, such that the skew is larger than the width of a pulse, then the two edges skew together, causing a changing signal value to occur. When this directive is on, the register and latch models recognize this case, and handle it as if the leading edge of the pulse was extended through the changing value. The directive

```
PULSE_FILTER ON;
```

activates the pulse filter. If unspecified, this directive is OFF.

SET_MIN_DELAYS

This directive affects the DELAY, RISE, and FALL properties in timing models. Because it affects delays internal to models, it should be used with caution. It allows all minimum delays above the specified value to be set to that value. The directive

```
SET_MIN_DELAYS 5.0;
```

sets all minimum delays greater than 5.0 nsec to 5.0 nsec. If unspecified, this directive is turned off.

TIMING_SIM_MODE

This directive causes the Verifier to verify for a set length of time instead of for a clock cycle. It is suitable for circuits that require verification over more than a single clock cycle. The syntax for this directive is:

```
TIMING_SIM_MODE ON;
```

The default value for this directive is OFF.

Notes on usage:

1. The length of time in nanoseconds for which you want to verify is specified with the CLOCK_PERIOD directive.

2. Assertion statements must be changed as required. For example, with a clock assertion !C 0-5, and these directives:

   CLOCK_PERIOD 100;

   CLOCK_INTERVALS 10;

   the clock is asserted for 50 ns and deasserted for 50 ns. With these directives:

   CLOCK_PERIOD 500;

   CLOCK_INTERVALS 10;

   TIMING_SIM_MODE ON;

   the value of CLOCK_PERIOD is changed to 500 but this does NOT automatically change the assertion statement. This gives you a clock that is asserted for 250 ns and deasserted for 250 ns.

   You also need to change the assertion statement to:

   !C 0-1, 2-3, 4-5,6-7, 8-9

   or its more compact equivalent:

   !2C 0-1

Using this directive can greatly improve execution speed for circuits containing large amounts of feedback. Error detection may not be as complete when this directive is used.

TS_BUS_TYPE

The Timing Verifier has two modes for simulating tri-state buffers and buses: the true tri-state function and a modified function called DOT_OR. The tri-state DOT_OR function is not identical to the wire gate DOT_OR function. The true tri-state function only gives useful output signal values when 0/1 behavior is specified for the enable signal. The modified tri-state function is less conservative and is used for designs with stable/changing behavior on the enables of the tri-state driver.

The TS_BUS_TYPE directive specifies which tri-state function is used for all tri-state logic in the design. The default value of the directive is DOT_TS. The directive

```
TS_BUS_TYPE DOT_OR;
```

models TS gates as activated when the ENABLE signal is STABLE. The directive

```
TS_BUS_TYPE DOT_TS;
```

models TS gates as activated only when 0/1 behavior is given for the ENABLE signal.

The truth tables for the logic functions performed by the TS BUS and the TS BUF for the two modes, and for the TS type wire gates, are described in the section on Primitives later in this manual.

If this directive is unspecified, tri-state buses are modeled DOT_TS.

SECTION 4
TIMING ASSERTIONS

In order for the Timing Verifier to function properly you need to specify the periodic behavior of the clock signals in the design. This is done by attaching an assertion statement to each clock signal. These assertion statements, or clock assertions, are prefixed with either !C or !P.

A second type of timing assertion that can be attached to signals are the !S and !D assertion statements. These assertion statements are used for non-clock signals and specify the signal behavior as either stable or changing.

Both types of assertion statements can be added in GED to a signal name on a drawing, or they can be entered in the case file. This section explains both types of timing assertions and the use of the case file.

4.1 CLOCK ASSERTIONS AND SIGNAL ASSERTIONS

Clock assertions are preceded with either !C or !P. These assertions specify when during the clock period the signal is asserted, and when not. They describe the clock signal as 0/1 behavior. Because clocks often have common skew, clock skew can be added to the calculation. The only difference between the !C and the !P assertion statements is the amount of skew specified. If all of the clocks in a design have the same amount of skew, then the !P assertion statement is not needed.

For complete timing verification it is also often necessary to specify the periodic behavior of primary input signals. In the Subtractor circuit shown in Figure 4-1, A<3 . . 0> and B<3 . . 0> are interface signals sent from outside of this design. If you do not specify their periodic behavior with an assertion statement, the Timing Verifier sets their values to stable at the beginning of verification. This allows the

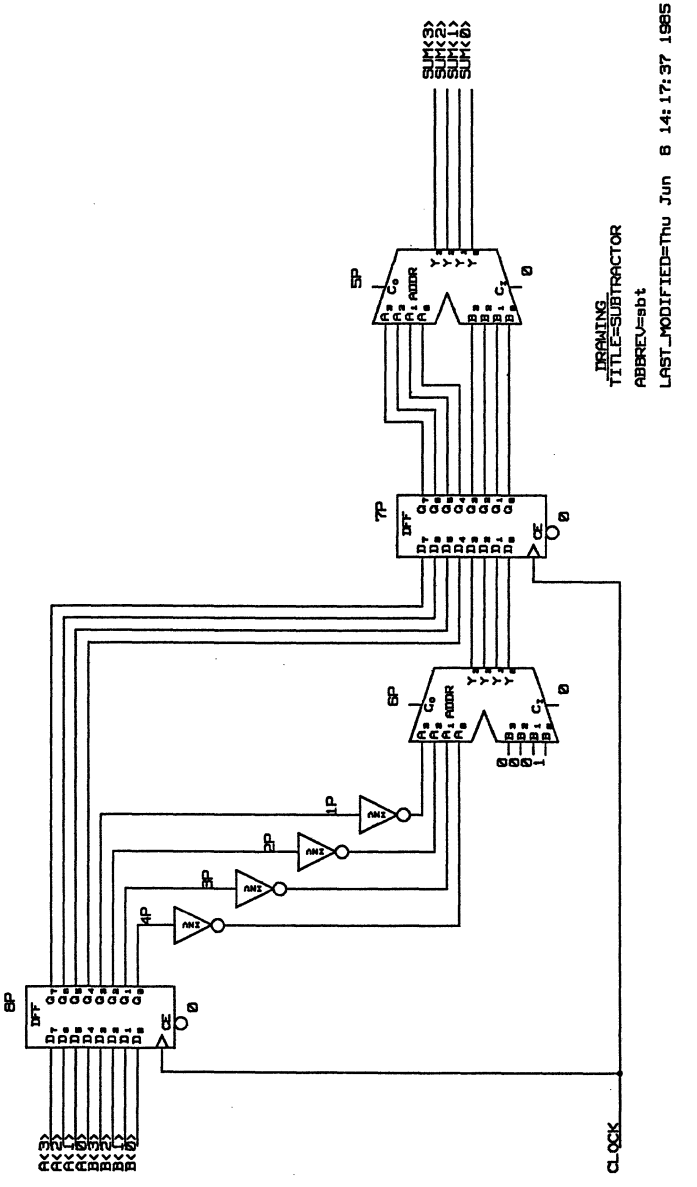

Figure 4-1. Subtractor Circuit

Timing Verifier to check the propagation delay through the first flip-flop, and all of the other timing delays for the rest of the circuit. But it does not allow the Timing Verifier to check the setup time or hold time for the first flip-flop. You can only ascertain whether these two time requirements for the first flip-flop are met by specifying the periodic behavior of the interface signals. Notice, however, that the behavior of the interface signals does not need to be specified in as much detail as does that of the clock signal. The Timing Verifier only needs to know if the signals A<3 . . 0> and B<3 . . 0> are STABLE during the setup time required by the flip-flop. It does not need to know whether the signal value at that time is 0 or 1.

Because of this difference between clock assertion statements and assertion statements for other signals, the prefixes !S and !D are used for assertion statements for signals other than clocks. The !S and !D assertion statements only specify the signal behavior in terms of stable and changing. More details on the syntax are given later in this section.

Another related use of the !S and !D assertion statements is to verify that output signals that are interface signals to other circuitry perform within the expected time ranges. In the Subtractor circuit shown in Figure 4-1 an assertion statement could be added to the signal SUM<3 . . 0>. The Timing Verifier would then report whether or not the calculated behavior of that signal agreed with the assertion statement. Although this check is not absolutely necessary because the data could be extracted from the signal history of the signal SUM<3 . . 0>, it is a feature that greatly facilitates the debugging of complex designs.

TYPES OF SIGNAL ASSERTIONS

Timing assertion statements may be one of four different types, each of which bears a unique prefix: !C, !P, !S, !D. The general form of an assertion statement is an assertion prefix followed by a time specifier. Timing assertions are either added to signal names or placed in the case file (case.dat). The same syntax is used for timing assertions in either location. This section gives examples of timing

assertions in signal names. For the use of timing assertions in the Case file (case.dat), see the section, Using the Case File, below.

The four types of signal assertions are discussed in order below.

1. C -- This prefix is used to specify the 0,1 behavior of a clock signal. The signal is asserted during the time specified by the time specifier. Signals having an !C time assertion that does not include skew as part of the assertion receive the clock skew specified with the CLOCK_SKEW directive in the Verifier command file. Clock skew that is specified in the timing assertion overrides skew specified with the CLOCK_SKEW directive. An !C assertion can be used on any signal for which you need to specify precise 0,1 behavior.

2. P -- This prefix is used to specify the 0,1 behavior of a precision clock signal. The !P assertion is identical to the !C assertion except that the signal using !P receives the skew specified by the PREC_CLOCK_SKEW directive. The !P assertion provides a way to make two groups of clock signals and assign to each group a different amount of skew. The skew assigned to signals having an !P assertion need not be more precise than that assigned to signals having an !C assertion.

3. S -- This prefix is used to specify the stable, changing behavior of a signal. The signal is defined to be STABLE during the time specified by the time specifier. Skew may be added as part of the timing assertion. If no skew is given, then the time specifier is assumed to be exact.

   When an !S assertion is added to a signal that is not a primary input, the assertion statement specifies an initial value for the signal. If, during the course of verification, the computed value for the signal differs from the specified value, the computed value overrides. The signal name appears in the listing file

under the heading "Signals not meeting their assertion specifications".

4. D -- This prefix is used to specify the stable, changing behavior of a signal. This assertion is the same as the !S assertion except that the signal value specified with an !D assertion statement is NEVER overridden by a computed value during the course of verification. The signal name appears in the listing file under the heading "Signals not meeting their assertion specifications". The use of an !D assertion on signals in feed back paths that are broken by latches can significantly speed up the execution of the Timing Verifier.

ADVANCED USE OF ASSERTIONS

Timing assertions can be used to create abstract models. An abstract model of a part P consists of a body drawing and an abstract timing model. The abstract timing model is constructed solely of buffers -- all input signals are received by buffers and the output signals driven by buffers. The output of each receiver buffer has a local signal with a timing assertion on it matching the input timing specification of the corresponding pin of P. The input of each drive buffer has a local signal with a timing assertion on it matching the output timing specification of corresponding output pin of P. This approach can be expanded to have small amounts of logic in the abstract model to achieve more complex logic or timing behavior as required.

4.2 TIMING ASSERTIONS IN SIGNAL NAMES

Signal assertions are added to the name portion of a signal name. They must be placed in the signal name after the name string, and before the bit subscript. The examples below illustrate the correct placement of an assertion statement within a signal name:

1. CLOCK A !C 0-4
2. ALU$DATA !C 0-4 <15 . . 3>
3. CLK!C0-4*
4. CLOCK A !C 1-2
5. DATA B !D 1-2 \WD 2.0-3.5

These examples display the following general rules:

Spaces are optional.

The timing assertion is part of the name string. This means that items 1 and 4 above are different signals because they have different signal names.

Bit subscripts, the assertion character, and properties go after a timing assertion.

Do not confuse a timing assertion with the assertion character. The assertion character (* is the default) indicates that a signal is low asserted. A timing assertion specifies WHEN a signal is asserted, or when a signal is stable. It does NOT specify when a signal is high. For example, item 3 in the list above is a low asserted signal that is low for the first four intervals of the period. For a clock with a period of 100 ns and 10 intervals, the waveform representation of this signal is shown in Figure 4-2:

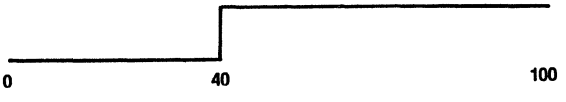

Figure 4-2. Low Asserted Signal

The syntax for signal assertions is as follows:

! *sub-interval assertion prefix time specifier skew*

Of these, only the assertion prefix and the time specifier are required. The four assertion prefixes are described above. The syntax for time specifiers, sub-intervals, and skew are described below.

TIME SPECIFIERS

The detailed syntax is:

time specifier = *time interval, time interval ...*

Where *time interval* is a positive integer or decimal number. A time specifier is one or several time intervals separated by commas. Here are examples of the three types of time intervals.

A. specifies a single clock interval. The signal is asserted or stable during this interval. Example: !C 4

B. specifies a range of clock intervals. The signal is asserted or stable during this range of clock intervals. Example: !C 0-4

C. specifies a clock interval followed by a time in nanoseconds. The signal becomes asserted at the beginning of the interval and stays asserted for the specified number of nanoseconds. Example: !C 2+ 3.2

All three types define the signal behavior in terms of the clock intervals specified with the CLOCK_INTERVALS directive in the Verifier command file. Several examples of each type appear below. For these examples, the clock period is 100 ns, and there are eight clock intervals, each 12.5 ns long. CLOCK_SKEW and PREC_CLOCK_SKEW are both set to 0.

Type A: Single Clock Interval

This form specifies a pulse whose width is one interval long.

CLK!P 4-5	a signal that is high from t4 to t5. This is the 5th interval. 0:0.0, 1:50.0, 0:62.5
CLK!P4(-2,5)	the same signal with asymmetrical skew added. 0:0.0, R:48.0, 1:55.0, F:60.5, 0:67.5

Notice that !P4-5 and !P4 both specify the clock interval that starts at t4. This is the 5th interval. The 1st clock interval starts at t0.

CLK!C 2,5*	a signal that is low for one interval starting at t2, and low again for one interval starting at t5. This signal is low for the 3rd and 6th intervals. 1:0.0, 0:25.0, 1:37.5, 0:62.5, 1:75.0
CLK!P2.2,5.7	a signal that is high for 1 interval (12.5 ns) starting at t2.2, and high for 1 interval starting at t5.7. 0:0.0, 1:27.5, 0:40.0, 1:71.3, 0:83.8

Notice that decimal numbers are permitted in time specifiers and in skews. The Timing Verifier rounds off numbers to 3 decimal places for calculations. Signal history can be reported at a resolution of 0 - 3 decimal digits. The OUTPUT_RESOLUTION directive selects this resolution. This type of time specifier is used for a pulse whose width scales with the clock intervals, but whose starting time does not.

Type B: Range of Intervals

This type specifies a range of intervals, or a series of ranges of intervals.

CLK!P 0-2	a signal that is high during the first 2 intervals. 1:0.0, 0:25.0
CLK!P0,4,7-8	a signal that is high during the 1st, 5th, and 8th intervals. 1:0.0, 0:12.5, 1:37.5, 0:50.0, 1:87.5

Notice that the signal is continually high from 87.5ns through 12.5 ns of the next clock cycle.

SIG!S1-4,6-7.3	a signal that is stable from t1 to t4, then changing, then stable for 1.3 intervals starting at t6. C:0.0, S:12.5, C:50.0, S:75.0, C:91.3
SIG!S2-4(-1,1)	a signal that is stable for the 3rd and 4th intervals with symmetrical skew of 1 ns. C:0.0, S:26.0, C:49.0

Notice that the first number in the parentheses is the negative skew, the negative sign is required.

Type C: Partial Clock Interval

This type specifies a pulse with a start time that is specified in terms of clock intervals, and a width specified in nanoseconds. It is used to specify pulses whose widths do not scale with the clock intervals.

SIG !S2+ 11.3	a signal that goes stable at t2 and is stable for the next 11.3 ns. C:0.0, S:25.0, C:36.3

-CLK!P3+ 7.0,4+ 9.0 a signal that goes low at t3 and stays low for 7 ns, then goes low again at t4 for 9 ns.
1:0.0, 0:37.5, 1:44.5, 0:50.0, 1:59.0

PRECEDING A SIGNAL ASSERTION WITH A SUB-INTERVAL

The *sub-interval* is a positive integer that must divide evenly into the number of intervals specified with the CLOCK_INTERVALS directive in the Verifier command file. The sub-interval must also be greater than or equal to the clock intervals specified in the time specifier. Here is an example for a clock with a period of 100ns and 50 intervals (of 2 ns each):

!10 C 0-8

The meaning of this timing assertion can be paraphrased as:

In every block of 10 intervals, go high for the first 8 intervals.

This produces a signal that is high for 16 ns, low for 4 ns, and then repeats this pattern 5 times in the clock period. The waveform representation of this signal is shown in Figure 4-3:

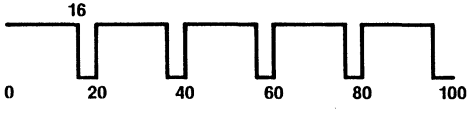

Figure 4-3. Clock with Sub-intervals

The subinterval must divide evenly into the number of intervals in the clock period so that the repeating pattern is regular for every clock cycle.

The time specifier cannot specify an interval greater than the subinterval. That is,

!10 C 8-10

is a legal time assertion that can be paraphrased as:

In every block of 10 intervals, go high for the intervals 8 through 10.

But the following example is NOT permitted and causes a syntax error:

!5 C 8-10

because a block of 5 intervals does not contain the intervals 8 through 10.

ADDING SKEW TO A SIGNAL ASSERTION

Skew can be added to any signal assertion on an individual basis. When skew is added to an !C or !P assertion, it overrides the skew specified in the CLOCK_SKEW and PREC_CLOCK_SKEW directives.

Clock skew added to signal assertions is calculated in the same manner for rising and falling edges. Skew may be specified as symmetrical or asymmetrical around the edge. (Notice that skew added with the CLOCK_SKEW and PREC_CLOCK_SKEW directives is always symmetrical.) The syntax for clock skew in a timing assertion is

(*-negative skew, positive skew*)

A negative sign must precede the negative skew. Here is an example of a timing assertion with skew:

CLOCKA !C 0-1 (-1.1834, 1)

R:0.0, 1:1.0, F:8.8, 0:11.0, R:98.8

The positive skew causes the clock to rise at 0 ns, and to go high at 1.0 ns. The negative skew is rounded off to 1.183 (remember the Timing Verifier calculates in picoseconds) and then is rounded to 1.2 for output (according to the

value of the OUTPUT_RESOLUTION directive). The clock is therefore reported as falling at 8.8 ns, and rising at 98.8 ns.

4.3 USING THE CASE FILE

The case file (case.dat) is an optional input file to the Timing Verifier. See above for the name of the case file under different operating systems. Whenever a case.dat file is present in the directory where verification is run, that is, in the same directory as the verifier.cmd file, the Verifier uses the case file as input.

CASE FILE SYNTAX

Each line in the case file contains on the left a signal name, and on the right a signal value or timing assertion. The last line of the case file is

```
END.
```

Each line in the case file (except the last line) ends with a comma or a semicolon. A comma is used at the end of lines within a single case. A semicolon is used to end a case. The Case File does not distinguish between upper and lower case letters.

The signal name portion of the line includes the name portion of the signal name and a bit subscript (when appropriate). Properties included in a signal name (by using a text macro) are not included in the case file. Remember that timing assertions and signal class are both part of the name portion of a signal name. Therefore, they are both included as part of the signal name in the case file.

Signal names that contain spaces need to be enclosed in quotes in the case file. The bit subscript portion of the signal name is not enclosed in the quotes. See the SCALD Language Reference Manual for details on signal names.

For a design including the signals:

```
DATA \WD 3.0-4.0
CLOCK !C 0-4
SUM <3 . . 0>
```

here is a possible case.dat file:

```
DATA = '1',
'CLOCK !C 0-4' = '!P 1-5',
'SUM' <3..2> = '0';
```

Notice that the signal DATA is entered in the case file without properties and that it requires no quotes.

Notice that the timing assertion for the signal CLOCK is included as part of the signal name and that it is enclosed in quotes.

Notice that the bit subscript for the signal SUM is not included in the quotes, and that it may specify a subrange of bits. This lets you specify different values for different subranges of bits of a bus. In addition, low asserted signals are always specified as

```
-SIGNAL
```

and not as

```
SIGNAL*
```

CASE ANALYSIS

Some digital systems are designed to take advantage of data dependent delays. In such systems, worst case timing analysis is not appropriate because although certain very long data paths through the system exist, these paths are never used. In order to tailor the timing analysis appropriately for such systems, the Timing Verifier includes a mechanism called **case analysis**. To use case analysis, you enter timing assertions and signal values into the case file (case.dat). When the assertions and signal values that you enter into the case file all describe concurrent events, they

are said to describe "one case". To best verify systems with data dependent delays, multiple cases are entered in the case file. When multiple cases are entered in the case file, the circuit is verified once for each case specified.

A case specifies a list of signals, and for each signal a signal value or timing assertion. Only the signal values 0, 1, and S can be used in the case file.

Consider the following circuit:

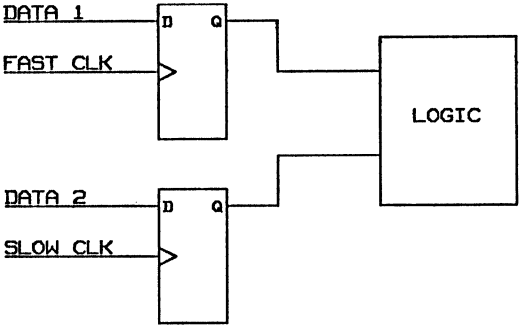

Figure 4-4. Case Analysis Example

Suppose that there are hundreds of transitions on FAST CLK for every transition on SLOW CLK. This could be modeled by using a very long period for analysis (the period of SLOW CLK) and specifying the full behavior of FAST CLK over that period. This is time consuming and produces much redundant information.

A better way to handle this kind of circuit is to consider that there are only two cases of interest. The first case is when SLOW CLK remains stable during an entire period of FAST CLK (i.e., only the upper register is being clocked). The second case is when both registers are being clocked.

To do this, a clock assertion is used on FAST CLK:

```
FAST CLK !C 0-5
```

With a 100 ns clock period that is divided into 10 clock intervals of 10 ns each, this produces a signal history of:

```
1:0.0, 0:50.0
```

If the behavior of SLOW CLK is unspecified and undriven, then the signal history of SLOW CLK is stable throughout the period of analysis. The two cases of interest may be specified as:

```
'SLOW CLK' = '!S 0-10';

'SLOW CLK' = '!C 0-5';
```

These two cases can be specified in the case file. Below is an example of a case file showing multiple cases.

Example Case File

Assume that a design contains the signals SELECT A1, SELECT B2*, CLOCK_RATE and DATA<3 . . 0>*. Then a sample case file including five different cases is:

```
'SELECT A1'  = '0';
;
'SELECT A1'  = '1',
'-SELECT B2'  = '0',
CLOCK_RATE   = '1';
-DATA = '0';
'SELECT A1' = '1',
'-DATA'<3 . . 0> = '1';
eND.
```

Notice that in the case file no distinction exists between upper case and lower case letters. See, for example, the last line in the sample file above.

The case file specifies five cases are to be run. It specifies no timing assertions.

1. The first case sets SELECT A1 to 0 for the intervals that this signal is stable (not changing).

2. The second case does not alter any of the signals. It is equivalent to running the Timing Verifier with no case file.

3. The third case sets SELECT A1 to 1, SELECT B2* to 0 and CLOCK_RATE to 1 for the intervals these signals are stable.

4. The fourth case sets the bus DATA<3 . . 0>* to 0 for the intervals it is stable.

5. The last case sets the bus DATA<3 . . 0>* to 1 and sets SELECT A1 to 1 for the intervals it is stable.

TIMING ASSERTIONS IN THE CASE FILE

Timing assertions can be added to signals as part of the signal name (using GED) or they can be entered in the case file. When signal assertions are entered in the case file, they can be changed without recompiling the drawing. Because recompilation can be time consuming, it is recommended that you enter timing assertions in the case file when performing the following tasks:

1. Trying several different timing assertions on a signal or group of signals.

2. Optimizing the performance of interface signals.

3. Determining the correct resetting sequence for a circuit.

4. Performing timing tests that require frequent changes to timing assertions.

Associating a timing assertion with a signal in the case file is identical to associating the assertion with the signal on the print. Subintervals and skews may also be included. See above under Timing Assertions in Signal Names for

the syntax for specifying subintervals and skews.

Below is an example of a case file showing timing assertions.

Example Case File

For a design with the signals, RESET*, INIT!C 0-3, DATA<15 . . 0> and ID!S 2-4<3 . . 0> in it, here is a case file that includes timing assertions:

```
-RESET = '0',
'INIT!C 0-3' = '!P 2-4(-2,+5)',
DATA<15 . . 0> = '!S 14-17,19';
'-RESET' = '!C 0-4',
'ID!S 2-4' = '!D 2-4';
END.
```

This case file describes two different cases. The first case is described by the first three lines of the file; the second case is described by the next two lines. In the first case, the signal RESET* is set to the value 0, and the signal INIT is given a different timing assertion. Notice that the timing assertion !C 0-3 is included as part of the signal name in the left part of the entry. This is because the timing assertion was entered on the drawing and therefore is part of the signal name. In this case the signal DATA is also given a timing assertion.

In the second case, the signal -RESET is given a timing assertion. Because -RESET is a low-asserted signal, the timing assertion !C 0-4 means that the signal is LOW for the first four intervals. The bit subscript for the signal ID !S 2-4 is not included. When a bit subscript is omitted, the value or timing assertion is assigned to the entire bus.

SECTION 5
DELAYS

Accurate timing verification requires not only the specification of component delays (as are provided in timing models for all Valid library parts), but also the specification of interconnect delays occurring in the design. Interconnect delays are technology dependent and design specific. They include both delays caused by wire length, and load-dependent delays. Because accurate interconnect delay data is only available late in the design cycle, after the design has been packaged and sent to a physical design system, the Timing Verifier provides several ways to approximate interconnect delay data in early design phases, and also provides a mechanism for feeding back detailed interconnect delay data from a physical design system.

The Timing Verifier also has a mechanism for specifying delays for tuned and gated clocks. This mechanism is Evaluation Directives and is described later in this section.

The Timing Verifier has four mechanisms for specifying wire delay information. Three of these are used in early design phases to estimate interconnect delays. The fourth is used to feed back delay information from the physical design system to the Timing Verifier.

- **Wire Delay File (delay.dat)** When accurate delay data is available from a physical design system, this data can be formatted (with the help of the Wire Delay Interface) into a file (delay.dat) that the Timing Verifier reads as an input file. In this file, a list element associates a delay with an **input** pin. Thus the delay on each stub of a signal that drives multiple loads may be specified.

- **Delay Estimator** For technologies where delay is load dependent, the Timing Verifier can use its Delay Estimator to calculate an estimated delay based on the number of loads and the size of the loads.

- **Wire Delay Property** Wire delay can be added to a signal by including the text macro \WD as part of the signal name, or by attaching the WIRE_DELAY property to the signal or pin. When this method is used, the specified delay is added only to that particular instance of the signal. If the signal has a synonym elsewhere in the design the delay is NOT added to the synonymed signal.

- **Wire Delay Directive** The Timing Verifier Wire Delay directive adds the specified delay to all wires in the circuit that connect to the input pin of a body and do not have a Wire Delay property attached.

The last two mechanisms in this list are designed to work together. Delay specified with the Wire_Delay Property overrides delay specified in the Wire_Delay Directive. All other combined uses of these four mechanisms causes the resultant delays to be added to each other. Since each of the first two mechanisms in this list are most frequently used as the only source of interconnect delay data, directives are provided to disable alternate sources of interconnect delay data.

5.1 WIRE DELAY DIRECTIVE

The WIRE_DELAY directive is used to specify a global delay or range of delays. The specified delay is attached to all signals in a design that are attached to the input pin of a library part and that do not have a delay property attached to them. See under Verifier Directives for additional information.

5.2 DELAY PROPERTIES

The Timing Verifier recognizes several delay properties. Of these, two are for use on logic drawings and the remainder are reserved for library development. The two delay properties that can be used on logic drawings are WIRE_DELAY and CLOCK_DELAY. Of these, the WIRE_DELAY property is by far the most common. Both properties can be attached to a signal or an input pin.

The delay properties used in library development are: DELAY, RISE, FALL, and CHIP_DELAY.

Verifier delay properties indicate that the signal is to be delayed by the time indicated with respect to the signal source.

Most delay properties (except RISE and FALL) can take two ranges of values. When two ranges are specified the first range specifies a min/max rise delay and the second a min/max fall delay. When a single range of values is specified, the rising and falling delays are assumed to be the same. The property value can also be a single fixed point number.

The syntax for the WIRE_DELAY and CLOCK_DELAY properties is the same. Here is an example:

WIRE_DELAY 2.0-5.6, 2.5-6.2

Here is a brief description of the delay properties recognized by the Verifier:

WIRE_DELAY — This is the most common wire delay property. The delay specified with this property overrides the delay specified with the WIRE_DELAY directive. The delay specified refers to the entire instance of the net, but not to other instances having the same name. All WIRE_DELAY properties in a design may be ignored by using the USE_DRAWING_WD

OFF directive. WIRE_DELAY is the only delay property affected by this directive. An individual instance of the WIRE_DELAY property may be set to zero with an evaluation directive.

CLOCK_DELAY — This delay is not affected by any evaluation directive, nor can it be ignored by using the USE_DRAWING_WD OFF directive. Its primary use is to describe a tuned clock which is adjusted to have some delay with respect to another logical version of the clock: See Evaluation Directives for additional details.

CHIP_DELAY — This delay is used primarily in library development to assign delay to an entire timing model. An evaluation directive can be used to set it to zero.

DELAY — This delay is used in library development to assign delay to signals in a timing model. The property is often attached to the BUF, REG, REG RS, LATCH, and LATCH RS primitives.

RISE — This delay property is used in library development to assign rise delay to a primitive in a timing model. This property can take only a single range of values, or a fixed point number.

FALL — This delay property is used in library development to assign fall delay to a primitive in a timing model. This property can take only a single range of values, or a fixed point number.

TEXT MACROS FOR DELAY PROPERTIES

To shorten signals and increase readability, three text macros have been predefined to specify delay properties. These are "WD", "CD", and "CKD" for wire delay, chip delay and clock delay respectively. The user may of course use either the full property name or the associated text macro.

ATTACHING A DELAY PROPERTY TO A SIGNAL OR A PIN

All Verifier delay properties can be attached to a signal or a pin. The delay is applied at each input pin to which the wire with the delay property (or signal name containing the delay property) is attached. For example consider the following example drawing with a D-type flip-flop and two inverters:

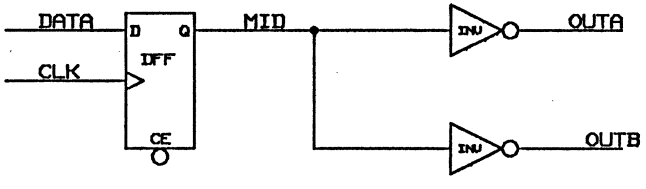

Figure 5-1. Using Delay Properties

The periodic behavior of the clock and data signals is specified in the case.dat file:

```
CLK = '!C 1-4',
DATA = '!S 0-2, 7-10';
```

Assume that the WIRE_DELAY directive has a value of 0. Notice that MID is the name of the signal that goes to both inverters.

The propagation delays for the flip-flop are RISE (8.5 - 27.0), FALL (9.0 - 27.0). The propagation delays for the inverters are RISE (4.5 - 15.0), FALL (5.0 - 15.0).

When the delay property is attached to the signal MID (as a signal property), the signals OUTA and OUTB both act identically to each other. With rise/fall analysis on, and a delay property of 2.0 ns, the signal value of MID is

S:0.0, C:18.5, S:37.0

and the signal values of OUTA and OUTB are

S:0.0, C:25.5, S:54.0

The 2.0 ns delay is not reflected in the signal history of MID, but only in that of OUTA and OUTB. Wire delay is always reflected in signal history after a lag. That is because all signal history reflects the behavior of the signal at the beginning of that signal. The signal MID begins at the output pin of the flip-flop. 8.5 ns is used for the propagation delay because it is the worse case possibility.

OUTA and OUTB start changing at 25.5 ns which is 5 + 2 ns after MID starts changing. The Verifier uses 5 ns and not 4.5 ns because it is doing rise/fall analysis. The 2 ns is the delay specified with the WIRE_DELAY property.

WIRE_DELAY AS A PIN PROPERTY

As an alternative to using the WIRE_DELAY property as a signal property, the WIRE_DELAY property may also be used as a pin property.

The WIRE_DELAY property may be attached to any input pin. The Timing Verifier does not recognize the WIRE_DELAY property when it is attached to an output pin.

In the circuit shown in Figure 5-1 the signal MID connects to two different input pins, one for each of the inverters. In this situation, when the delay is greater along one path than along the other, the WIRE_DELAY property can be attached to the input pin of one of the inverters, instead of to the entire signal. As a result, the delay affects one of the two output signals, but not both.

When the WIRE_DELAY property is attached to the input pin of the topmost inverter, the delay appears only in the signal history of OUTA and not of OUTB. Here is the signal history for both signals:

OUTA	S:0.0, C:25.5, S:54.0
OUTB	S:0.0, C:23.5, S:52.0

Delay properties are handled this way so that systems where delays are different on different "stubs" of a net may be modeled correctly.

5.3 DELAY ESTIMATOR

In many technologies, the time required for the output of a component to reach its loads is affected by both the interconnect delay and the size of the load:

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

Tdr is the component's rise delay

Kr is a device-specific constant related to changes in the output's rise time as a function of component loading

Tir is the rising edge delay due to wires

Tdf is the component's fall delay

Kf is a device-specific constant related to changes in the outputs fall time as a function of component loading

Tif is the falling edge delay due to wires

Note that Tdr, Tdf, Kr, Kf are device-specific; (load on the net) is net specific; and Tir, Tif are input specific. All quantities can assume both a min and max value.

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

When the signal delay due to loading effects is small, it is often adequate to estimate the delay and add it to the signal in GED using the WIRE_DELAY property, or to specify a global default value for delays by using the WIRE_DELAY directive.

When the loading effect is not small, the Timing Verifier has a more accurate Delay Estimator that takes into account static load, and provides a wire delay estimate based on the number of **stops** (inputs and outputs) on the net. This delay estimate is added to the basic component delay (specified with the RISE, FALL, and DELAY properties in the timing model).

To use the Delay Estimator you need to use the directive DELAY_ESTIMATOR ON and set the other appropriate directives to the required values. The Delay Estimator uses the following equation:

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

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

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

INTERACTION OF WIRE DELAYS WITH DELAY ESTIMATOR

The delays calculated by the delay estimator are used to adjust the delays of the driving components, and as such will be added to wire delays specified in the drawings or fed back wire delays specified in the wire delay file. The directive USE_DRAWING_WD allows the user to control whether the wire delays specified in the drawings will be used or not. In general, any fed back wire delays will override any wire delays specified in the drawings. It is suggested that USE_DRAWING_WD normally be turned off when the Delay Estimator is on.

COMPUTING NET DEPENDENT DELAYS

The Timing Verifier estimates the net delay on each net in six steps:

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

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

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

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

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

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

Counting Loads

Counting the inputs and outputs on a net is complicated by the presence of TIMES properties and dots. When a net is connected to the output pin or input pin of a library part that has the TIMES property attached to it, the value of the TIMES property affects the number of loads on the net.

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

> Each input pin is counted *n* times when TIMES = *n* and the sum of the values of the TIMES properties for each pin is used. The output pin to which the net is connected is counted once and when the output pin is connected to a library part having a TIMES property, the total number of inputs on the net is divided by *n*.

For example, a net is connected to an output pin of a library part with the property TIMES=3, and to three input pins. The input pins are on three different library parts that have, respectively, the properties TIMES=5, TIMES=2, and TIMES = 1. The load is calculated as 5 + 2 + 1 = 8 inputs on the net. This total number of inputs is divided by the value of the TIMES property of the output pin. This gives 8 divided by 3.

When a net is connected to several output pins that are dotted together (connected together) and the output pins are on library parts that have the TIMES property attached to them, the following rules apply:

The number of output pins on the net is the sum of each output pin ignoring the TIMES properties. The number of inputs on the net is counted as before, then divided by the smallest value of the TIMES property of any library part with an output on the net.

If phantom gates are used, they are collapsed to an explicit dot for the counting procedure.

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

Estimating Wire Delays

To estimate wire delay, the number of stops on each net is counted. If phantom gates are used, they are collapsed into an explicit dot for the stop counting process. The number of stops is converted to equivalent loads by table lookup using a table specified with the Timing Verifier directive WIRE_ESTIMATE. WIRE_ESTIMATE takes an argument list of fixed point numbers and an optional FAMILY specification. A net with *j* stops receives a wire delay

estimate given by the *j*th number in the list. The family specification allows for a number of different WIRE_ESTIMATE tables to be used in the same Timing Verification run. If a FAMILY body property is given on a primitive, then the WIRE_ESTIMATE table with the same FAMILY specification will be used. If no FAMILY body property is given on a primitive, then the WIRE_ESTIMATE table without a FAMILY specification will be used. An example set of WIRE_ESTIMATE directives are given below:

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

Computing Load Dependent Net Delays

In timing models, each primitive that drives an output pin of the part being modeled can have an optional body property, DRIVE. This property takes a pair of fixed point ranges as a value, the first range is the driver's Kr factor, the second its Kf factor. A range is required to specify both a minimum and maximum value. If no DRIVE property is specified, Kr and Kf are set to the default value specified by the DEFAULT_DRIVE directive. When neither DRIVE properties nor the DEFAULT_DRIVE directive are used, Kr and Kf are set to 0-0. If only one fixed point range or number is given, Kr and Kf are both set to that value.

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

USING THE DELAY ESTIMATOR

To use the delay estimator, follow these steps:

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

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

   - By attaching the DRIVE body property to each Timing Verifier primitive whose output is to display load-dependent behavior. See Appendix A for the DRIVE property syntax.

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

3. Specify pin loading. This is done by attaching the LOAD_FACTOR property to a pin of the Timing Verifier primitive that connects to the interface signal that represents the pin of the part whose load is to be specified. The LOAD_FACTOR property takes a fixed point number as its value. If no LOAD_FACTOR property is specified, a default LOAD_FACTOR value of 1 is used.

4. Specify a conversion table from stops to load equivalents using the WIRE_ESTIMATE directive. See under Directives for details.

Assuring Correct Load Counting

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

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

number of pieces equal to the smallest value of the TIMES properties found on the drivers.

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

In order to ensure correct counting of loads, timing models are constructed so that only one primitive (not counting checker bodies) is connected to each interface signal. The BUF primitive is used to accomplish this task.

WIRE DELAY FILE

An optional input file to the Timing Verifier is the wire delay file. This file associates delays with input pins of the parts in the system. When the Wire Delay file is used, the specified delay is applied to the signal driving the input pin before the component is simulated. This allows a delay to be associated with each stub on a net. This file is typically generated by feeding the wire delays calculated by a physical design system through the WIRE DELAY interface.

The Wire Delay file consists of a list of signal names and under each one, a list of the path names of the parts that the signal drives and a delay for each pin. Here is an example:

```
'SIGNAL 1' <5..0> :
    '(SYS ALU MUX)' = '2.3-3.4',
    '(SYS REG)' = '0.2-1.7,0.1-1.2';

'SIGNAL 2' <7..5> :
    '(SYS SHIFTER)' = '0.0-1.1';

'SIGNAL 2' <4..1> :
    '(SYS SHIFTER)' = '0.5-3.1';
```

```
'SIGNAL 2' <0> :
    '(SYS SHIFTER)' = '0.5-3.1';
end.
```

If no bit subscript is included as part of a signal name, the specified delay applies to all of the bits of a bus. To simplify feeding back of wire delays, a path name in the stop delay list may be a unique left substring of the actual path name. For Wire Delay File syntax see Appendix A.

INTERACTION OF WIRE DELAY FILE AND OTHER WIRE DELAYS

When a delay.dat file is available, the delay values in that file are ADDED to the wire delays specified by the three other methods: with the WIRE_DELAY property on the drawing, with the WIRE_DELAY directive, and with the Delay Estimator.

If the user has access to a delay.dat file, generated by the physical design system, the information in that file is probably more accurate than the delay information specified in the Directives file, on the drawing, or calculated by the Delay Estimator.

To use the delay.dat file as the only source of delay information, specify these directives:

```
DELAY_ESTIMATOR OFF;
USE_DRAWING_WD OFF;
WIRE_DELAY 0.0-0.0;
```

5.4 EVALUATION DIRECTIVES

Some high-speed digital systems use clocks that are tuned to compensate for delays in the system. A tuned clock is adjusted so that its timing behavior is independent of circuit delays. Other systems use gated clocks where the system only functions correctly when the gating signal properly "envelopes" the clock for all variations in circuit delays. In both cases, the Timing Verifier needs additional information in order to correctly verify the circuit. Evaluation

directives are used to direct the Timing Verifier to correctly evaluate circuits using tuned and gated clocks.

An additional evaluation directive is used to initialize an individual signal to a particular signal value.

Six evaluation directives are recognized by the Timing Verifier. Five are used with tuned and gated clocks, the sixth is used to initialize signals to a specified value. Here is a brief list of the function of each of the six evaluation directives:

V initializes the signal to a known value. Most often used to set a signal to 0, 1, or S (stable).

W sets the minimum delay of the wire to zero and subtracts the minimum delay from the maximum delay.

Z sets the wire delay and the gate delay to zero.

A checks that the non-clock input to a gate is stable when the clock input is enabling the gate. Directs the Timing Verifier to ignore all the inputs to the gate except the one with the A evaluation directive.

H sets the wire delay and the gate delay to zero and checks that the non-clock input to a gate is stable when the clock input is enabling the gate.

I directs the Timing Verifier to ignore all the inputs to the gate except the one with the I assertion. The output of the gate is simply the input signal (including any timing assertion) delayed by the propagation delay of the gate. This directive may be used on any type of gate.

The \E V evaluation directive may be used on any signal. All of the other evaluation directives may be applied to only one input of a gate.

ATTACHING AN EVALUATION DIRECTIVE TO A SIGNAL

The most convenient to attach an evaluation directive to a signal is to use the reserved text macro \E *n*, where *n* is one of the five letters V, W, Z, A, or H. The evaluation directives may also be specified in signal names by using the full form of the text macro: **\EVAL='I'**. This full form is required for the I evaluation directive when including it in a signal name.

The PROPERTY command can also be used to attach an evaluation directive to a wire. In this case, the property name is EVAL and the property value is V, W, Z, A, H, or I.

EVALUATION DIRECTIVE FOR SIGNAL INITIALIZATION

The \E V evaluation directive is used to initialize any signal to a value other than U (unknown). When this directive is not used, signals are initialized to U. Legal values for the \E V directive are:

\E V 0	initialize to 0
\E V 1	initialize to 1
\E V S	initialize to stable
\E V C	initialize to changing
\E V Z	initialize to high impedance
\E V U	initialize to unknown

A signal cannot be initialized to Rising or Falling. The values 0, 1, and S are the most useful values for this evaluation directive.

EXAMPLE CIRCUIT

Figure 5-2 shows a flip-flop with a gated clock signal. This circuit serves well to demonstrate the effect of the remaining five evaluation directives.

The circuit is verified first with no evaluation directive and then once with each of the five directives: **\E W, \E Z, \E H, \E A,** and **\E I.** The following Verifier directives file is used in all cases:

```
CLOCK_PERIOD 300.0;
CLOCK_INTERVALS 10;
TIMING_DIAGRAMS ON;
CLOCK_SKEW 0.0;
PREC_CLOCK_SKEW 0.0;
WIRE_DELAY 0.0-0.0;
MAX_ERRORS 50;
END.
```

The pertinent timing information for the LS08 and the LS74 follows:

LS08	Rise time delay : 4.0 - 15.0 ns Fall time delay : 5.0 - 20.0 ns
LS74	Rise time delay : 6.5-25.0 ns Fall time delay : 12.5-40.0 ns

In the case file, the input to the LS74 is asserted with a changing value from 210-240 ns. In a true design, this input would be generated by the circuit and would not have to be specified. The case.dat file is as follows:

```
'EN' = 1,
'INPUT' = '!S 0-7, 8-10';
END.
```

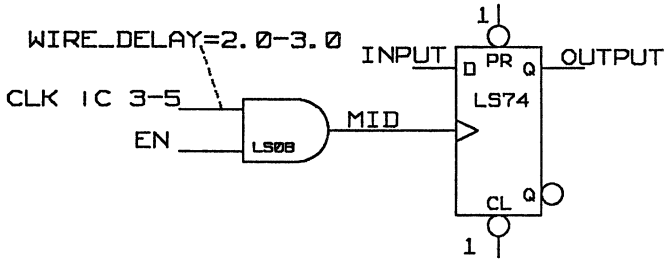

Figure 5-2. No Evaluation Directive

The signal history for the verification with no evaluation directive is:

CLK !C 3-5	0:0.0, 1:90.0, 0:150.0
EN	1:0.0
INPUT	S:0.0, C:210.0, S:240.0
MID	0:0.0, R:96.0, 1:108.0, F:157.0, 0:173.0
OUTPUT	S:0.0, C:102.5, S:148.0

EN is held high, so MID reflects CLK, its wire delay, and the component delay of the LS08. After CLK goes high at 90.0 ns, MID rises at 96.0 ns (90.0 from CLK + 4.0 min component rise delay + 2.0 min wire delay = 96.0), and is high at 108.0 ns (90.0 from CLK + 15.0 max component rise delay + 3.0 max wire delay = 108.0). CLK falls to zero at 150.0 ns causing MID to fall at 157.0 ns (150.0 + 5.0 min component fall delay + 2.0 min wire delay = 157.0), and MID to be stable at 173.0 ns (150.0 + 20.0 max component fall delay + 3.0 max wire delay = 173.0). The timing diagram for the circuit with no evaluation directives is shown in Figure 5-3.

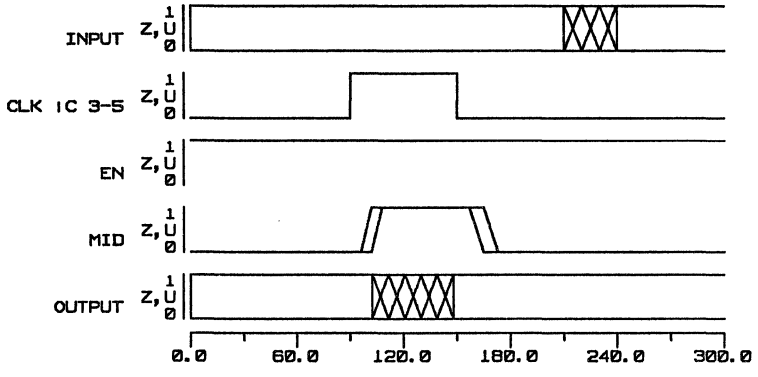

Figure 5-3. Timing Diagram: No Evaluation Directive

EVALUATION DIRECTIVES FOR CLOCK TUNING

The \E W and \E Z evaluation directives are used for tuned clocks.

\E W

This evaluation directive sets the minimum delay of the wire to zero and subtracts the minimum wire delay from the maximum wire delay. This is used for tuned clocks to cancel out wire delays. Figure 5-4 shows the circuit with the **\E W** evaluation directive and Figure 5-5 is the circuit's timing diagram.

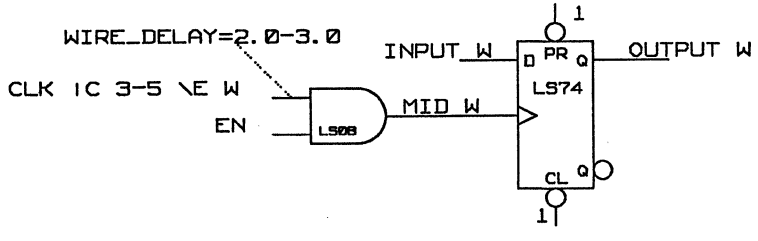

Figure 5-4. \E W Evaluation Directive

The signal history for the verification with the **\E W** evaluation directive is:

CLK !C 3-5	0:0.0, 1:90.0, 0:150.0
EN	1:0.0
INPUT W	S:0.0, C:210.0, S:240.0
MID W	0:0.0, R:94.0, 1:106.0, F:155.0, 0:171.0
OUTPUT W	S:0.0, C:100.5, S:146.0

The WIRE_DELAY properties are affected by this evaluation directive and are recalculated as follows: the minimum wire delay is set to zero, and the minimum wire delay is subtracted from the maximum wire delay leaving a recalculated wire delay of 1.0 ns (3.0 - 2.0 = 1.0). EN is held high so MIDW reflects CLK, the recalculated wire delay, and the component delay. CLK goes high at 90.0 ns causing MIDW to rise at 94.0 ns (90.0 + 4.0 min component rise delay + 0.0 minimum wire delay = 94.0) and MIDW to be high at 106.0 ns (90.0 + 15.0 max component rise delay + 1.0 recalculated wire delay = 106.0).

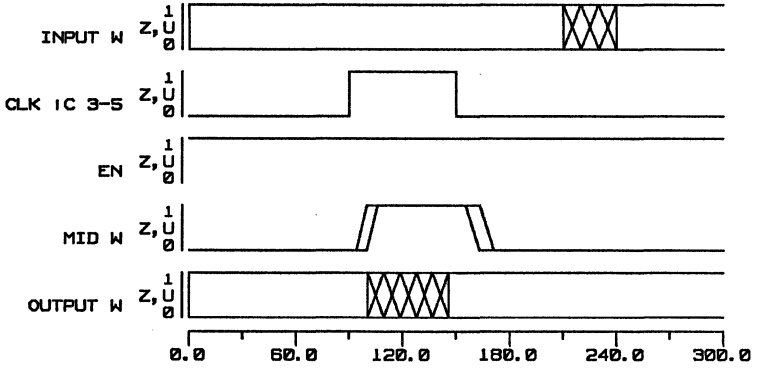

Figure 5-5. Timing Diagram: \E W Evaluation Directive

\E Z

This directive sets both the wire delay and the gate delay to zero. This is also used for tuned clocks. The following delay properties are affected by this evaluation directive:

WIRE_DELAY
DELAY
RISE
FALL
CHIP_DELAY

Since the component delay from the LS08 is not reflected and EN is held high, MIDZ changes with CLK. The circuit is shown in Figure 5-6, and Figure 5-7 shows the timing diagram.

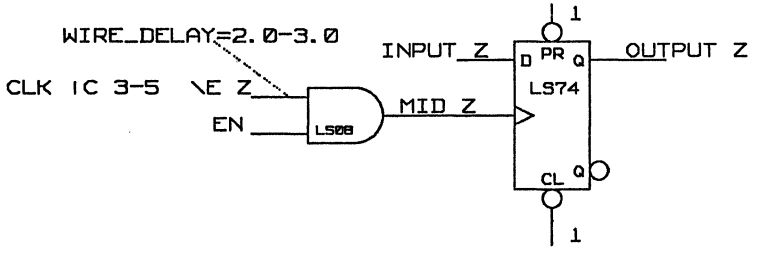

Figure 5-6. \E Z Evaluation Directive

The signal history for the verification with the \E Z evaluation directive is:

CLK !C 3-5	0:0.0, 1:90.0, 0:150.0
EN	1:0.0
INPUT Z	S:0.0, C:210.0, S:240.0
MID Z	0:0.0, 1:90.0, 0:150.0
OUTPUT Z	S:0.0, C:96.5, S:130.0

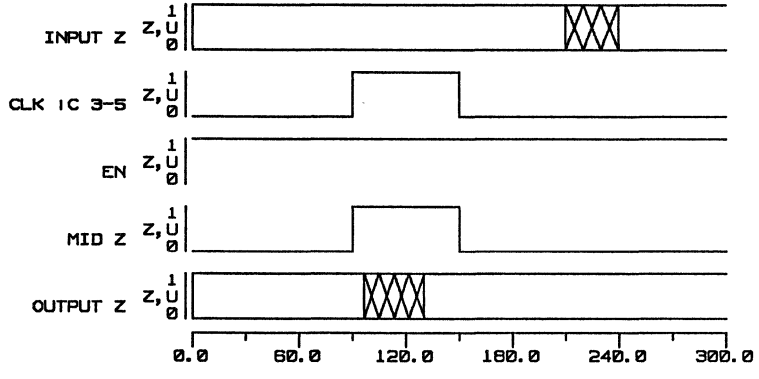

Figure 5-7. Timing Diagram: \E Z Evaluation Directive

EVALUATION DIRECTIVES FOR CLOCK GATING

A clock can be gated using either an AND gate or an OR gate. In our example, the clock is gated with an AND gate. Correct performance of the digital system requires that the gating signal (in this case, EN) be stable while the clock signal is in its asserted state. EN must be stable when CLK !C 3-5 is high. When an OR gate is used instead of the LS08, EN must be stable when CLK !C 3-5 is low.

\E A

This directive is used on AND and OR gates to check that the non-clock input is stable when it is enabling the gate. The A directive may be used only on AND and OR gates where one input of the gate is driven by a clock signal (a signal with a 'C' or 'P' assertion). When the 'A' directive is used, the output of the gate (MID A) is calculated as a function of CLK, any wire delay, and the propagation delay through the LS08. MID A does not reflect the EN signal.

None of the delay properties are affected by this evaluation directive.

For this example, EN is specified (in the **case.dat** file) to be a pulsing signal. Because the LS08 is an AND gate, EN is controlling the gate when CLK is asserted (high). An error occurs if EN changes while CLK is asserted. If an OR gate is used, the EN signal controls the gate when CLK is low and an error occurs if EN changes while CLK is low.

Figure 5-8 shows the circuit with the \E A directive.

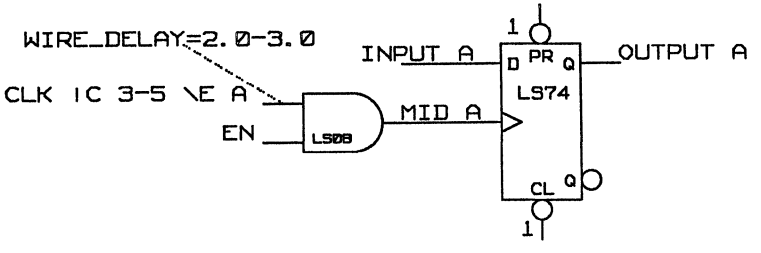

Figure 5-8. \E A Evaluation Directive

The signal history for the verification with the **\E A** evaluation directive is:

CLK !C 3-5	0:0.0, 1:90.0, 0:150.0
EN	1:0.0, 0:60.0, 1:120.0, 0:210.0
INPUT A	S:0.0, C:210.0, S:240.0
MID A	0:0.0, R:96.0, 1:108.0, F:157.0, 0:173.0
OUTPUT A	S:0.0, C:102.5, S:148.0

The timing diagram is shown is Figure 5-9.

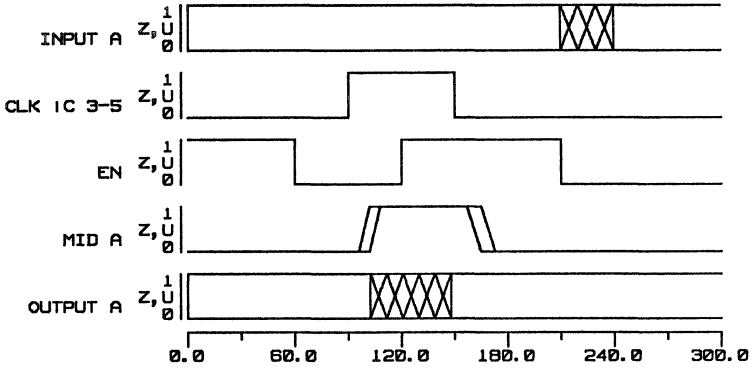

Figure 5-9. Timing Diagram: \E A Evaluation Directive

Since EN is changing while CLK is asserted, the following timing violation is generated:

```
#1 Timing error (166): Input changing while clock
      is asserted
Drawing name = LS08.TIME.1.1
     Path name to primitive is
     "(DRC LS08.1P.2AN1P)"
CK INPUT = CLK !C 3-5 \E A (0.0:0.0)
0:0.0, R:92.0, 1:93:0, F:152.0, 0:153.0
DATA INPUT = EN
1:0.0, 0:60.0, 1:120.0 0:210.0
```

TUNED AND GATED CLOCKS

When a clock is tuned and gated, you use either the 'H' directive or the 'I' directive. The 'H' directive is a variation of the 'A' directive. It causes the Timing Verifier to ignore any wire delay and the propagation delay through the LS08, and also to check that the non-clock input to the gate is stable when it is enabling the gate.

The 'I' directive causes the Timing Verifier to ignore all other inputs to the gate except the CLK input and to calculate MID I as a function of the CLK input, any wire delay, and the propagation delay through the LS08. An error is NOT generated if the non-clock input changes while it is controlling the gate.

\E H

This directive is similar to **\E A**. MIDH is a function of CLK only; it is not effected by EN, propagation delay, or wire delay.

The following delay properties are affected by this evaluation directive:

WIRE_DELAY
DELAY
RISE
FALL
CHIP_DELAY

Figure 5-10 shows the circuit using the **\E H** evaluation directive, and Figure 5-11 shows the timing diagram for the circuit.

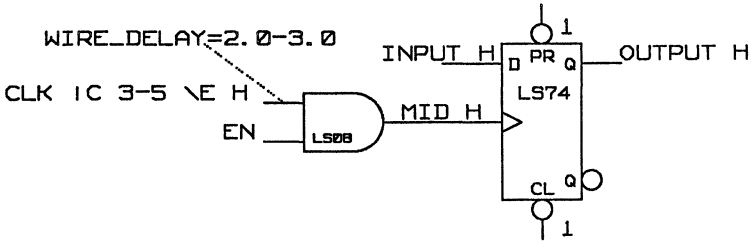

Figure 5-10. \E H Evaluation Directive

The signal history for the verification with the **\E H** evaluation directive is:

CLK !C 3-5	0:0.0, 1:90.0, 0:150.0
EN	1:0.0, 0:60.0, 1:120.0, 0:210.0
INPUT H	S:0.0, C:210.0, S:240.0
MID H	0:0.0, 1:120.0, 0:150.0
OUTPUT H	S:0.0, C:126.5, S:160.0

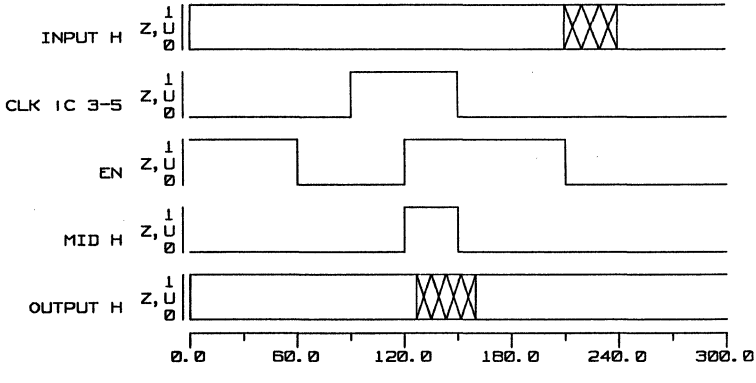

Figure 5-11. Timing Diagram: \E H Evaluation Directive

A timing error is generated because EN changes while the clock is asserted:

```
#1 Timing error (166): Input changing while clock
     is asserted
Drawing name = LS08.TIME.1.1
    Path name to primitive is
    "(HDR LS08.1P.2AN1P)"
CK INPUT = CLK !C 3-5 E H (2.0:3.0)
0:0.0, 1:90.0, 0:150.0
DATA INPUT = EN
1:0.0, 0:60.0, 1:120.0 0:210.0
```

\E I

This directive causes the Timing Verifier to ignore all inputs to the gate except the one with the **\E I** assertion. It is similar to **\E A** except it does not generate an error if the non-clock input changes while it is controlling the gate. The output of the gate is the input signal (with assertion) delayed by propagation delay of the gate and wire delay. This directive may be used on any gate type, but only one input to a gate may have an **\E I** assertion.

None of the delay properties are affected by this evaluation directive.

To use this evaluation directive you must use the long form of the text macro, like this:

```
\EVAL='I'
```

or the property name EVAL with the value 'I'. The shortened text macro \E I is not permitted.

The signal EN is specified as a stable/changing value in the **case.dat** file. Figure 5-12 shows the circuit with the \EVAL='I' evaluation directive, and Figure 5-13 shows the circuit's timing diagram.

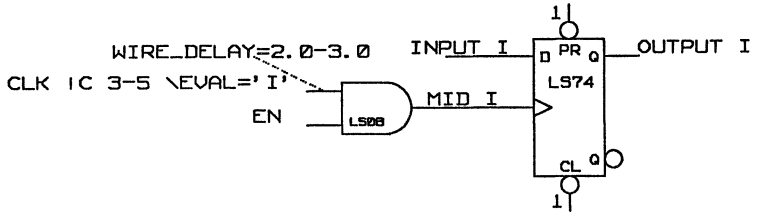

Figure 5-12. \EVAL='I' Evaluation Directive

The signal history for the verification with the **\EVAL='I'** evaluation directive is:

CLK !C 3-5	0:0.0, 1:90.0, 0:150.0
EN	C:0.0, S:60.0, C:120.0, S:210.0
INPUT I	S:0.0, C:210.0, S:240.0
MID I	0:0.0, R:96.0, 1:108.0, F:157.0, 0:173.0
OUTPUT I	S:0.0, C:102.5, S:148.0

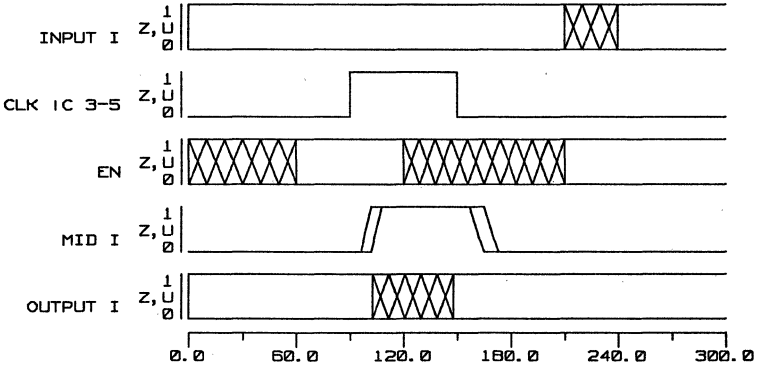

Figure 5-13. Timing Diagram: \EVAL='I'

In this example, if the **\EVAL='I'** evaluation directive is not used, a timing violation occurs, because the clock input to the LS74 has a changing value. The timing violation without the **\EVAL='I'** directive is:

```
#1 Timing error(159): Minimum pulse width
timing violation
Input is driven with changing value at time = 96.0
Minimum HIGH = 25.0, Minimum LOW = 15.0
Drawing name = LS74.TIME.1.1
     Path name to primitive is
     "(DRC LS74.2P MPW5P)"
CK INPUT = MID I (+ 10.0)
0:0.0, R:96:0, S:98:0, C:124.0, 0:163.0
```

The signal history for the circuit without the **\EVAL='I'** directive is:

CLK !C 3-5	0:0.0, 1:90.0, 0:150.0
EN	C:0.0, S:60.0, C:120.0, S:210.0
INPUT I	S:0.0, C:210.0, S:240.0
MID I	0:0.0, R:96.0, S:108.0, C:124.0, 0:173.0
OUTPUT I	S:0.0, C:102.5, S:213.0

If EN is specified in terms of ones and zeros as in the examples with the **\E A** and the **\E H** directives, and the **\EVAL='I'** directive is not used, the minimum pulse width error does not occur. If the EN signals in the **\E A** and **\E H** examples are specified as changing/stable, a minimum pulse width error is generated instead of the error:

"Signal changing while clock is asserted."

The timing diagram is shown in Figure 5-14.

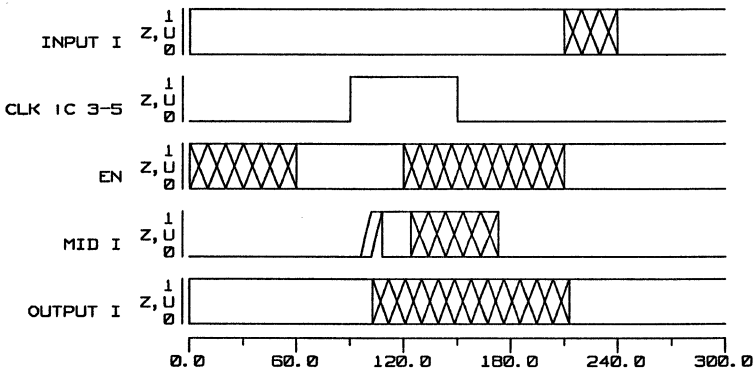

Figure 5-14. Timing Diagram Without EVAL = 'I'

DEFINING COMPLEX TUNED AND GATED CLOCKS

The **\E H, \E W, \E Z, \E A,** and **\E I** evaluation directives can be made to ripple through multiple gates by specifying them repeatedly as shown in the example in Figure 5-15.

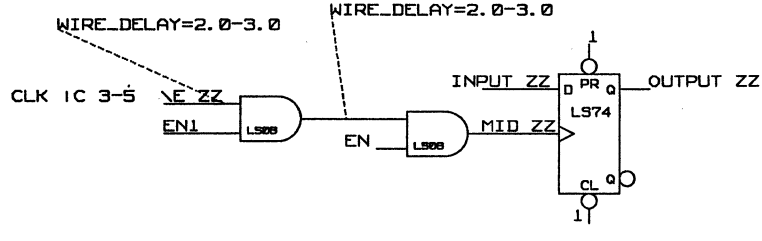

Figure 5-15. \E='ZZ' Evaluation Directive

The **\E ZZ** causes the wire and component delay from both of the LS08s to be canceled out before reaching the LS74. The signal values and timing diagrams for this example are the same as for the **\E Z** example. The signal history is shown below:

CLK !C 3-5	0:0.0, 1:90.0, 0:150.0
EN	1:0.0
EN1	1:0.0
INPUT ZZ	S:0.0, C:210.0, S:240.0
MID ZZ	0:0.0, 1:90.0, 0:150.0
OUTPUT ZZ	S:0.0, C:96.5, S:130.0

The W directive can be used with the Z directive to zero out the minimum wire delay of the LS08 output. You use the directive **\E=ZW**. The ZW causes the input wire delay and the propagation delay of the LS08 to be set to zero (Z evaluation directive). It also causes the minimum wire delay of the output wire MID ZW to be set to zero (W evaluation directive).

EVALUATION DIRECTIVES USED IN MULTILEVEL COMPONENTS

When a component is defined with multiple levels of primitives, it is desired that the evaluation directives refer to the entire path through the component, rather than to a single primitive that the component is made up of. If the component definition is a single level drawing, then the Timing Verifier automatically causes the evaluation directive string to count all of the primitives as one element. A user can also put the body property 'KEEPDIRECTIVE' on a primitive which will cause it to propagate the entire evaluation string through it, rather than taking the first evaluation letter off of it. This property is useful if a hierarchical definition for a component is used and the evaluation directives only want to increment once when going through the component.

SECTION 6
TIMING MODELS

As part of each Valid-supplied library, there is a timing model for each component in the library. The timing model models the timing behavior of the component and is used by the Timing Verifier. Timing models are built by library developers from a specific set of parts called Timing Verifier primitives. The model is entered into the SCALD-system as a GED drawing having the extension .TIME. For example, the timing model of an LS74 is a drawing with the name LS74.TIME. It can be viewed on the screen using the EDIT command. Permissions are usually set on component models so that only the librarian or root has permission to change the models.

Timing models are built with a dual purpose in mind: to accurately model the timing behavior of a component while keeping the model as simple as possible. When timing models are kept simple the Timing Verifier runs quickly and efficiently. Timing models therefore focus on timing characteristics and do not exhaustively simulate the logical behavior of the component.

A timing model needs to correctly model the delays of all signals through the component (propagation delay), and for clocked and complex components it needs to check setup and hold times, pulse width, and, when appropriate, edge to edge time.

To efficiently perform these various functions there are three types of time primitives.

6.1 TIME PRIMITIVES

Time Primitives are the building blocks of timing models. Each primitive is a logic function that can be fully described in a truth table and that is represented by a GED drawing. The time primitives make up the TIME library on your SCALDsystem. A timing model may contain only parts from this library and from the STANDARD library. (In rare cases a time model can also contain a hierarchical body that is made up solely of time primitives.) Time primitives were created to best describe timing functions. To this end there are three basic groups of time primitives.

The first group of time primitives is made up of elements based on familiar SSI and MSI functions like gates and flip-flops. The second group of time primitives is made up of non-standard functions that are particularly useful in timing models. The third group of time primitives is made up of checker primitives that are added to timing models to catch timing errors. They do not model circuit functionality. Each of these groups of time primitives is described more fully below. Truth tables for each primitive are given later in the section.

STANDARD FUNCTIONS

The first group of time primitives are based on familiar SSI and MSI components. They perform some, but not all, of the functions of these parts. In this group you find AND gates, OR gates, registers, latches, multiplexers, and so on. MSI components like latches, registers and multiplexers can be used as primitives for the Timing Verifier because the Verifier is only interested in their delay characteristics and not in their total functionality. Models for different latches, for example, are built by using one of the two latch primitives and additional SSI gates where needed and attaching delays to the latch primitive and to wires so that the propagation delays for each signal are correct and according to the data book.

The primitives in this group are:

AND	REG
OR	REG RS
XOR	2 MUX
LATCH	4 MUX
LATCH RS	8 MUX
TS BUF	

NON-STANDARD FUNCTIONS

The next group of time primitives are non-standard logic functions that are particularly suited to modeling timing functionality. Some of these primitives, like the Buffer and the Resistor, are similar to the familiar components, but may be used somewhat differently in a timing model. Other parts, such as the CHANGE primitive, were created specially for the Timing Verifier. These parts are used to attach delay properties to various parts of a model, provide accurate load calculations, and otherwise assure efficient and correct functioning of the model.

The primitives in this group are:

CHG	change gate
BUF	buffer
IDENTITY	identity
RES	resistor
THRESHOLD	threshold
TRANSMISSION GATE	transistor
UNI TRANS GATE	uni-directional transistor

The non-standard primitives are described more fully in Section 6.2.

CHECKER PRIMITIVES

The last category of time primitives differ from the other two categories because they do not model functionality. Instead, they are added to timing models of clocked components to check for setup and hold time violations, and other clock related errors. They are sometimes called checker primitives. These primitives are:

SETUP HOLD
SETUP RISE HOLD FALL
MIN PULSE WIDTH
EDGE TO EDGE

The error-checking primitives are described more fully in Section 6.3.

USING TIME PRIMITIVES

Of the three types of time primitives described above, the first group need little further explanation. Truth tables for each appear later in the chapter. Because the second and third groups of primitives are somewhat less familiar, a description of the function of each of these follows in the next two sections. First are descriptions of the non-standard primitives, followed by descriptions of the primitives used for error checking.

Bubbling of Primitive Pins

All Timing Verifier primitives have bubbleable pins. This feature allows negative edge triggering of latches, buffers to become inverters, etc.

Each input and output of every primitive may be "bubbled" independently. (See Graphics Editor, BUBBLE command.) When this is done, it is as if an inverting buffer were inserted between the signal (input or output) and the primitive itself. The characteristics of the primitive itself are not changed in any way. This is useful for creating inverting buffers (by bubbling the input or output of a BUF), nand gates, nor gates, negative edge triggered registers, etc.

The use of a bubbled input on a MIN PULSE WIDTH primitive is a good example of the statement that the primitive itself is unchanged. In order to check a low asserted signal (e.g., CLOCK *) to make sure that it is low for at least 20.0 ns, you may use a MIN PULSE WIDTH primitive with a bubbled input and a HIGH=20.0 property.

6.2 NON-STANDARD PRIMITIVES

THE CHANGE PRIMITIVE

The most important member of this group is the CHANGE primitive. The change primitive takes an input signal and tells you whether that signal is stable, changing, or unknown. Frequently this is all the information the Timing Verifier needs. When you add delay to the CHANGE primitive, you effectively model simple propagation delay through a component. To model, for example, the propagation delay from the A and B inputs to the sum (Y output) of an adder (let's say an LS283), you use the CHANGE primitive and attach the appropriate delays using the PROPERTY command. The model is very simple because the delay through this component is the same regardless of the values being added. Adding in the appropriate delay for CARRY IN complicates the model only slightly.

THE BUFFER AND IDENTITY PRIMITIVES

The Buffer primitive is used as a convenient place to attach delay properties in a model. It is also used to isolate outputs so that correct load calculations can be performed. As one would expect, the value of a signal is not changed by the Buffer primitive (with the exception of the value Z). Buffers are also used to isolate outputs for correct load checking.

The Identity primitive is a special case of the Buffer primitive. It retains the identity of all signal values including Z. It also retains the signal strength of all input signals (see Signal Strength, below).

THE RESISTOR PRIMITIVE

The Resistor primitive has the same truth table as the Identity primitive. But the Resistor primitive changes the strength of HARD input signals to SOFT signal strength. Since most other signal strengths in a design are HARD, this means that the value of the Resistor output can be overridden by a competing HARD value. This primitive is used to assure the correct modelling of circuits using pull-up resistors. For HARD and SOFT input strengths, the Resistor outputs a SOFT signal strength; for UNDRIVEN input strengths, the Resistor outputs an UNDRIVEN signal strength. For more information on signal strengths (HARD, SOFT, UNDRIVEN) see the section later in this chapter.

THE THRESHOLD PRIMITIVE

The THRESHOLD primitive has a threshold input and a single output pin. The primitive behaves somewhat like an input-state (0 or 1) detector whereby its output remains changing until its threshold input is asserted. This primitive is very seldom implemented.

THE TRANSMISSION GATE

The TRANSMISSION GATE primitive has an enable input EN, and two bi-directional pins T1 and T2. If the enable input is ZERO, then both T1 and T2 are set to high-impedance (Z). If EN is ONE, then T1 and T2 are tied together using the same function as the tri-state bus (TS BUS) defined below.

THE UNI TRANS GATE

The UNI TRANS GATE primitive is a uni-directional transistor. It has an enable input EN, an input pin I, and an output pin T. If the enable input is ZERO, then T is set to high-impedance (Z). If EN is ONE, then the value and strength of I is passed to T.

6.3 ERROR-CHECKING PRIMITIVES

SETUP HOLD

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

SETUP = 0.0
HOLD = 0.0

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

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

SETUP RISE HOLD FALL

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

SETUP = 0.0
HOLD = 0.0

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

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

EDGE TO EDGE

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

MIN = 0.0
MAX = 0.0

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

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

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

MIN PULSE WIDTH

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

LOW = 0.0
HIGH = 0.0

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

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

6.4 TRUTH TABLES FOR TIMING FUNCTIONS

The truth tables for each of the Timing Verifier primitives are given below. In the case where more than one entry applies to a given set of input conditions, the first entry takes precedence.

Truth tables are also given for three ancillary functions that are not represented by primitives. These three functions are:

the TS BUS (tri-state bus) function,
the SET RESET function,
the LATCH_ERR_MODEL function

The TS BUS function is used by the Timing Verifier when TS BUFs are wired together into a TS BUS. The SET RESET function is used to model the LATCH RS and the REG RS primitives. The LATCH_ERR_MODEL function is used in conjunction with a directive and the LATCH primitive to provide three behavioral models for a closing latch.

AND, OR, XOR, and CHANGE FUNCTIONS The truth tables for the AND, OR, XOR, and CHANGE(CHG) functions are given in the following tables:

AND	0	1	S	R	F	C	U	Z
0	0	0	0	0	0	0	0	0
1	0	1	S	R	F	C	U	U
S	0	S	S	R	F	C	U	U
R	0	R	R	R	C	C	U	U
F	0	F	F	C	F	C	U	U
C	0	C	C	C	C	C	U	U
U	0	U	U	U	U	U	U	U
Z	0	U	U	U	U	U	U	U

OR	0	1	S	R	F	C	U	Z
0	0	1	S	R	F	C	U	U
1	1	1	1	1	1	1	1	1
S	S	1	S	R	F	C	U	U
R	R	1	R	R	C	C	U	U
F	F	1	F	C	F	C	U	U
C	C	1	C	C	C	C	U	U
U	U	1	U	U	U	U	U	U
Z	U	1	U	U	U	U	U	U

XOR	0	1	S	R	F	C	U	Z
0	0	1	S	R	F	C	U	U
1	1	0	S	F	R	C	U	U
S	S	S	S	C	C	C	U	U
R	R	F	C	C	C	C	U	U
F	F	R	C	C	C	C	U	U
C	C	C	C	C	C	C	U	U
U	U	U	U	U	U	U	U	U
Z	U	U	U	U	U	U	U	U

CHG	0	1	S	R	F	C	U	Z
0	S	S	S	C	C	C	U	U
1	S	S	S	C	C	C	U	U
S	S	S	S	C	C	C	U	U
R	C	C	C	C	C	C	U	U
F	C	C	C	C	C	C	U	U
C	C	C	C	C	C	C	U	U
U	U	U	U	U	U	U	U	U
Z	U	U	U	U	U	U	U	U

TS BUF and TS BUS FUNCTIONS

The tri-state buffer primitive has two inputs: DATA and ENABLE. When the SIZE property is used on the TS BUF (for example SIZE = 2), the DATA input and output are size replicated, but the ENABLE is not. The ENABLE signal is common to both buffers.

By default, this primitive operates in "tri-state mode" as shown in the first table below. When the ENABLE is STABLE, the output is UNKNOWN. This is a conservative model of tri-state behavior. The alternative "wire-or mode" is less conservative and is provided to accommodate designs in which the ENABLE signal is specified as STABLE/CHANGING. Selection of the "wire-or mode" or the "tri-state mode" is controlled by the TS_BUS_TYPE directive.

The directive TS_BUS_TYPE DOT_TS; selects tri-state mode and is the default. The directive TS_BUS_TYPE DOT_OR; selects wire-or mode. See the Directives section for additional explanation.

The truth table for the TS BUF primitive in the default tri-state mode is given below.

ENABLE INPUT

DATA INPUT	TS BUF	0	1	S	R	F	C	U	Z
	0	Z	0	U	C	C	C	U	U
	1	Z	1	U	C	C	C	U	U
	S	Z	S	U	C	C	C	U	U
	R	Z	R	U	C	C	C	U	U
	F	Z	F	U	C	C	C	U	U
	C	Z	C	U	C	C	C	U	U
	U	Z	U	U	U	U	U	U	U
	Z	Z	U	U	U	U	U	U	U

TRI-STATE MODE

The truth table for the TS BUF primitive in the alternate wire-or mode is given below.

ENABLE INPUT

DATA INPUT	TS BUF	0	1	S	R	F	C	U	Z
	0	Z	0	0	C	C	C	U	U
	1	Z	1	1	C	C	C	U	U
	S	Z	S	S	C	C	C	U	U
	R	Z	R	R	C	C	C	U	U
	F	Z	F	F	C	C	C	U	U
	C	Z	C	C	C	C	C	U	U
	U	Z	U	U	U	U	U	U	U
	Z	Z	U	U	U	U	U	U	U

WIRE-OR MODE

TS BUS

When the outputs of two or more TS BUFs are tied together you have what we call a tri-state bus (TS BUS) as shown below. The TS BUS is a special kind of primitive because it is not represented by a GED drawing. You cannot add a TS BUS to a timing model.

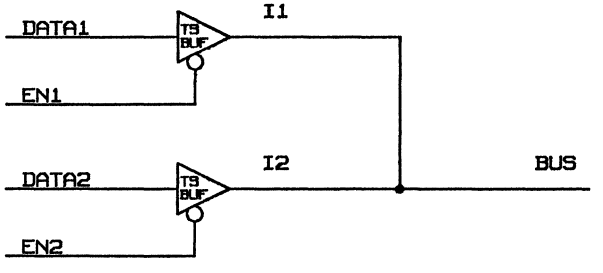

Figure 6-1. TS BUS

Because the drivers are tri-state and share the use of the BUS by means of separate ENABLE signals, the logical function represented here is not the same as that of a wire-gate. With a tri-state bus, the only two meaningful configurations are:

1. Only one TS BUF is enabled at a time.

2. If two are enabled, they carry identical output signal values.

Almost all other conditions produce the signal value U (unknown) on the BUS. See the tri-state mode table below.

The Timing Verifier evaluates this circuitry in accordance with the tables below. The TS BUS, like the TS BUF primitive, operates by default in "tri-state mode" but can also operate in "wire-or mode". The selection of tri-state or wire-or mode is made with the TS_BUS_TYPE directive and is made globally for an entire design. See above under TS BUF and see also TS_BUS_TYPE in the Directives section for further information. The truth table for the TS BUS function in the default tri-state mode is given below.

		I1							
	TS BUS	**0**	**1**	**S**	**R**	**F**	**C**	**U**	**Z**
	0	0	U	U	U	F	U	U	0
	1	U	1	U	R	U	U	U	1
	S	U	U	U	U	U	U	U	S
I2	**R**	U	R	U	R	U	U	U	R
	F	F	U	U	U	F	U	U	F
	C	U	U	U	U	U	U	U	C
	U	U	U	U	U	U	U	U	U
	Z	0	1	S	R	F	C	U	Z

TRI-STATE MODE

The truth table for the TS BUS function in the alternate wire-or mode is given below.

		I1							
	TS BUS	**0**	**1**	**S**	**R**	**F**	**C**	**U**	**Z**
	0	0	S	S	R	F	C	U	0
	1	S	1	S	R	F	C	U	1
	S	S	S	S	C	C	C	U	S
I2	**R**	R	R	C	R	C	C	U	R
	F	F	F	C	C	F	C	U	F
	C	C	C	C	C	C	C	U	C
	U	U	U	U	U	U	U	U	U
	Z	0	1	S	R	F	C	U	Z

WIRE-OR MODE

BUF AND THRESHOLD PRIMITIVES

The truth tables for the BUF and THRESHOLD primitives are given in the following tables:

BUF	
INPUT	OUTPUT
0	0
1	1
S	S
R	R
F	F
C	C
U	U
Z	U

THRESHOLD	
INPUT	OUTPUT
0	C
1	1
S	C
R	C
F	C
C	C
U	U
Z	U

RES AND IDENTITY FUNCTIONS

Below are the truth tables for the RES and IDENTITY primitives. See also Signal Strength in Section 7.

RES	
INPUT	OUTPUT
0	0
1	1
S	S
R	R
F	F
C	C
U	U
Z	Z

IDENTITY	
INPUT	OUTPUT
0	0
1	1
S	S
R	R
F	F
C	C
U	U
Z	Z

LATCH PRIMITIVE

The LATCH primitive is affected by the value of the TRANSITION property and the value of the LATCH_ERR_MODEL directive. Both of these features are described below.

The LATCH primitive has a DATA and an ENABLE input. When ENABLE is bubbled the inverse of the chart applies. In the chart **X** represents any signal value and <> means "not equal to".

LATCH			
EN	**LAST OUTPUT**	**DATA**	**OUTPUT**
0	0,1,S	X	= 0,1,S
0	R,F,C,U,Z	X	S
1	X	0,1,S,R,F,C	= DATA
1	X	U,Z	U
R	= DATA	0,1,U,Z	0,1,U,U
R	S	S	S **
R	= DATA	all other cond.	C
R	<> DATA	U,Z	U
R	0	1,S	R
R	1	0,S	F
R	R,F,C,U,Z	0,1,S	C
R	R,1	R	R
R	F,0	F	F
R	all other	cond.	C

** Note:If there has been no DATA transition since EN was last 1 or R and the latch is being simulated SMOOTH.

LATCH (cont'd)			
EN	**LAST OUTPUT**	**DATA**	**OUTPUT**
F	= DATA	0,1,S,U,Z	0,1,S,U,U
F	= DATA	R,F,C	C
F	X	U,Z	U,U
F	0	1,S	R
F	1	0,S	F
F	C	0,1,S	= DATA
F	R,1	R	R
F	F,0	R	F
F	all other	cond.	C
S	= DATA	X	= DATA
S	<> DATA	0,1,S	S
S	1	R	R
S	0	F	F
S	<> 1	R	C
S	<> 0	F	C
S	X	C	C
S	all other	cond.	U
C	X	U,Z	U
C	= DATA	0,1,S,R,F,C	= DATA
C	all other	cond.	C
Z	X	X	U
U	X	X	U

If the INPUT undergoes a transition while the latch is closing, then a setup/hold time violation has occurred. Under these conditions the latch is modeled in one of three ways depending on the value of the LATCH_ERR_MODEL directive.

The LATCH_ERR_MODEL Directive

When the ENABLE signal of a latch primitive has the value F (falling) and the DATA signal undergoes a transition, an error condition occurs. When this error condition occurs, signal values are calculated in accordance with the value of the LATCH_ERR_MODEL directive.

This directive takes one of three values: OPEN, CLOSED, or CONSERVATIVE. The default value is CONSERVATIVE. Truth tables are given below for each of the three values. Notice that in all cases the last value of the ENABLE signal was F (falling).

LATCH_ERR_MODEL = OPEN			
LAST EN	**LAST OUTPUT**	**DATA**	**OUTPUT**
F	X	U,Z	U
F	0	0,1,S	R
F	1	0,1,S	F
F	C	0,1,S	0,1,S
F	S,R,F,U,Z	0,1,S	C
F	R,1	R	R
F	F,0	F	F
F	all other	cond.	C

LATCH_ERR_MODEL = CLOSED			
LAST EN	**LAST OUTPUT**	**DATA**	**OUTPUT**
F	0,1,S	X	0,1,S
F	R,F,C,U,Z	X	S

LATCH_ERR_MODEL = CONSERVATIVE			
LAST EN	**LAST OUTPUT**	**DATA**	**OUTPUT**
F	X	U,Z	U
F	0	0,1,S	R
F	1	0,1,S	F
F	S,R,F,C,U,Z	0,1,S	C
F	R,1	R	R
F	F,0	F	F
F	all other	cond.	C

The LATCH primitive has by default the property TRANSITION = GLITCHY. This means that when the LATCH is clocked, the output of the LATCH always changes, even when the input remains stable. If the property TRANSITION = SMOOTH is attached to the LATCH, the output of the LATCH does not change when the LATCH is clocked and the input remains stable.

LATCH RS

The LATCH RS primitive is a LATCH primitive that also has asynchronous RESET and SET inputs. First the LATCH output is computed for the current input values, then the SET RESET function is applied to the outputs. The SET RESET function is different for TRANSITION = GLITCHY and TRANSITION = SMOOTH. The SET RESET function is described below.

TRANSITION PROPERTY

The transition property takes the values SMOOTH and GLITCHY and is used to provide accurate models of both glitchy and smooth registers and latches. When it is attached to the REGISTER and LATCH (and REG RS and

LATCH RS) primitives, it alters the functioning of the primitive to model either GLITCHY or SMOOTH behavior. The REGISTER and REG RS primitives each have two body VERSIONS, the first GLITCHY, the second SMOOTH. Select the appropriate version with the VERSION command in GED.

The LATCH and LATCH RS primitives have only one body VERSION. These primitives are GLITCHY by default. To change their behavior, attach the TRANSITION = SMOOTH body property. The truth tables for these primitives include information on both glitchy and smooth models.

The transition property can be attached to any Timing Verifier primitive, but it only affects the behavior of the REG, REG RS, LATCH, LATCH RS, and MUX primitives.

SET RESET FUNCTION

The SET RESET function is combined with the LATCH primitive and the REG primitive, respectively, to produce the LATCH RS and the REG RS functions. It is not directly accessible as a Timing Verifier Primitive. The SET RESET function is different for TRANSITION = SMOOTH and GLITCHY. The function inherits its TRANSITION property from the LATCH RS or REG RS of which it is a part.

Here are the truth tables for the SET/RESET function in glitchy mode and in smooth mode. Notice that certain values for NEW OUTPUT are given in terms of the CHG function. This is the function performed by the CHANGE primitive described above.

SET/RESET in GLITCHY MODE			
RESET	**SET**	**OUTPUT**	**NEW OUTPUT**
0	0	X	OLD OUTPUT
0	X	1	1
0	1	<> 1	1
0	R,F,C	<> 1	C
0	S	0,S	S
0	S	R,F,C	C
0	S	U,Z	U
0	U,Z	<>1	U
X	0	0	0
1	0	X	0
R,F,C	0	<> SET	C
U,Z	0	<> SET	U
S	0	1,S	S
S	0	R,F,C	C
S	0	U,Z	U
all	other	cond.	CHG (OUTPUT, RESET,SET)

CHG is the change function defined above.

SET/RESET in SMOOTH MODE			
RESET	**SET**	**OUTPUT**	**NEW OUTPUT**
0	0	X	OUTPUT
0	X	1	1
0	1	X	1
0	R	0	R
0	S	0,S	S
0	S	R,F,C	C
0	S	U,Z	U
0	F,C	X	C
0	U,Z	X	U
X	0	0	0
1	0	X	0
R	0	0	F
S	0	0,1,S	S
S	0	R,F,C	C
S	0	U,Z	U
F,C	0	X	C
U,Z	0	X	U
1,R	F	0,1,S	0,F,F
1,R	F	R,F,C	C
1,R	F	U,Z	U
F	1,R	0,1,S	R,1,R
F	1,R	R,F,C	C
F	1,R	U,Z	U

CHG is the change function defined above.

REG FUNCTION

The REG primitive implements a rising edge triggered register. The truth tables for the REG functions appear below. The tables are for different values for the CLOCK signal.

REG when CLOCK = 1			
LAST CK	**INPUT**	**LAST OUT**	**NEXT OUT**
0	0,1	0,1	LAST OUT
0	1,R	0,R	R
0	0,F	1,F	F
0	S	S	S**
1	X	<>0,1,S	S
1	X	0,1,S	LAST OUT
S	X	X	LAST OUT
R	0,1	S	LAST OUT
F	X	<>0,1,S	S
C,U,Z	INPUT =	LAST OUT	LAST OUT
C,U,Z	INPUT <>	LAST OUT	S

** Note: If the REG is SMOOTH and there were no input transitions.

REG when CLOCK = C or R			
LAST CLK	**INPUT**	**LAST OUT**	**NEXT OUT**
X	0,1	0,1	LAST OUT
X	1,R	0,R	R
X	0,F	1,F	F
X	S	S	S**

** Note: If the REG is SMOOTH and there were no input transitions.

REG when CLOCK = 0, S, or F			
LAST CLK	INPUT	LAST OUT	NEXT OUT
X	X	<>0,1,S	S
X	X	0,1,S	LAST OUT

REG when CLOCK = U or Z			
LAST CLK	INPUT	LAST OUT	NEXT OUT
X	X	X	U

When the REG primitive has the property TRANSITION = GLITCHY, and the REG is clocked, the output of the REG always changes, even when the input remains stable. When the REG primitive has the property TRANSITION = SMOOTH, the output of the REG does not change when the REG is clocked and the input remains stable.

REG RS

The REG RS primitive is the same as the REG except that it also has asynchronous RESET and SET inputs. First the REG output is computed for the current input values, then the SET RESET function is applied to the output.

THE 2, 4 AND 8 MUX FUNCTIONS

The 2 MUX, 4 MUX, and 8 MUX primitives implement 2-input, 4-input, and 8-input multiplexers. If any of the select inputs on these multiplexers has a known value of 0 or 1, then only the possibly selected data inputs will be looked at when calculating the output value. If more than one data input might be selected, the output value is calculated by using the CHANGE function on the set of selected data inputs.

If the MUX has no TRANSITION property or TRANSITION = GLITCHY, then any input transition causes an output transition of the appropriate slope. If TRANSITION = SMOOTH, then if the output state before and after an input transition is the same, there is no output transition.

SECTION 7
WIRE GATES

The Timing Verifier simulates multiple driven nets (buses) by inserting a gate in the network. All the drivers of the bus are reconnected to the gate's inputs. The output of the gate drives all the inputs on the bus. The type of gate inserted depends on the bus type. If the bus is a wire-or or a wire-and type bus, the inserted gate is a WIRE-OR gate or a WIRE-AND gate, respectively. If the bus is a tri-state bus, the inserted gate is a TS BUS functioning in either DOT_TS mode or DOT_OR mode depending on the value of the TS_BUS_TYPE directive.

The output of a Timing Verifier primitive may assume one of three strengths, HARD, SOFT or UNDRIVEN. Strengths are required to correctly model circuit nodes that have multiple drivers on them when those drivers (outputs) have different drive capabilities. A typical example of this is a tri-state bus that is pulled-up with a resistor. When none of the tri-state drivers are on, the bus is in the one state. When a single driver drives the bus to zero, the bus assumes the zero state. The use of signal strengths provides a way of modelling the fact that the resistor output is weaker than a bus driver output.

All primitives except the resistor, the identity function, and wire gates ignore the strengths of their input signals. The functions of the resistor and identity gate are described above.

By default, the output of all devices except RES, IDENTITY and wire gates is HARD. The output of a resistor primitive is SOFT, unless the input to the resistor is UNDRIVEN, then the output is UNDRIVEN. The output of the IDENTITY primitive is the same as its input. The output strength of a wire gate is the same as the strongest input strength.

WIRE GATE TRUTH TABLES

The functions of the wire gates are shown in the tables below. Different tables are provided for different signal strengths.

WIRE OR
HARD

HARD	0	1	S	R	F	C	U	Z
0	0	1	S	R	F	C	U	0
1	1	1	1	1	1	1	1	1
S	S	1	S	R	F	C	U	S
R	R	1	R	R	C	C	U	R
F	F	1	F	C	F	C	U	F
C	C	1	C	C	C	C	U	C
U	U	1	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

The WIRE OR truth table for HARD input strength is identical to the OR truth table for all values except Z. When one of the inputs to a WIRE OR gate is Z, the output takes the value of the other input.

WIRE OR
SOFT

HARD \	0	1	S	R	F	C	U	Z
0	0	1	S	R	F	C	U	0
1	1	1	1	1	1	1	1	1
S	S	1	S	R	F	C	U	S
R	R	1	R	R	R	R	U	R
F	F	1	F	F	F	F	U	F
C	C	1	C	C	C	C	U	C
U	U	1	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

The truth table for the HARD/SOFT WIRE OR gate is identical to the HARD/HARD WIRE OR truth table except for the values R and F.

WIRE OR
UNDRIVEN

HARD \	0	1	S	R	F	C	U	Z
0	0	1	S	R	F	C	U	0
1	1	1	1	1	1	1	1	1
S	S	S	S	S	S	S	U	S
R	R	1	R	R	R	R	R	R
F	F	1	F	F	F	F	F	F
C	C	C	C	C	C	C	C	C
U	U	U	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

The WIRE OR function for SOFT/HARD input signal strengths is obtained by transposing the values of the HARD/SOFT table given above.

The WIRE OR function for SOFT/SOFT input strength is identical to the HARD/HARD table given above.

The WIRE OR function for SOFT/UNDRIVEN input strength and UNDRIVEN/SOFT input strength is identical to the HARD/UNDRIVEN table given above.

The WIRE OR function for UNDRIVEN/HARD input strength is obtained by transposing the values of the HARD/UNDRIVEN table given above.

WIRE OR

UNDRIVEN	0	1	S	R	F	C	U	Z
0	0	S	S	S	S	S	U	0
1	S	1	S	S	S	S	U	1
S	S	S	S	S	S	S	U	S
R	S	S	S	S	S	S	U	S
F	S	S	S	S	S	S	U	S
C	S	S	S	S	S	S	U	S
U	U	U	U	U	U	U	U	U
Z	0	1	S	S	S	S	U	Z

WIRE AND
HARD

HARD	0	1	S	R	F	C	U	Z
0	0	0	0	0	0	0	0	0
1	0	1	S	R	F	C	U	1
S	0	S	S	R	F	C	U	S
R	0	R	R	R	C	C	U	R
F	0	F	F	C	F	C	U	F
C	0	C	C	C	C	C	U	C
U	0	U	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

The WIRE AND truth table for HARD input strength is identical to the AND truth table for all values except Z. When one of the inputs to a WIRE AND gate is Z, the output takes the value of the other input.

WIRE AND SOFT

HARD \	0	1	S	R	F	C	U	Z
0	0	0	0	0	0	0	0	0
1	0	1	S	R	F	C	U	1
S	0	S	S	R	S	S	U	S
R	0	R	R	R	C	C	U	R
F	0	F	F	C	F	C	U	F
C	0	C	C	C	F	C	U	C
U	0	U	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

WIRE AND UNDRIVEN

HARD \	0	1	S	R	F	C	U	Z
0	0	0	0	0	0	0	0	0
1	0	1	S	S	F	C	U	1
S	S	S	S	S	S	S	S	S
R	0	R	R	R	R	R	R	R
F	F	F	F	F	F	F	F	F
C	C	C	C	C	C	C	C	C
U	U	U	U	U	U	U	U	U
Z	0	1	S	S	F	C	U	Z

The WIRE AND function for SOFT/HARD input signal strengths is obtained by transposing the values of the HARD/SOFT table given above.

The WIRE AND function for SOFT/SOFT input strength is identical to the HARD/HARD table given above.

The WIRE AND function for SOFT/UNDRIVEN input strength is identical to the HARD/UNDRIVEN table given above.

The WIRE AND functions for UNDRIVEN/SOFT input strength and UNDRIVEN/HARD input strength are obtained by transposing the values for the HARD/UNDRIVEN table given above.

WIRE AND

UNDRIVEN	0	1	S	R	F	C	U	Z
0	S	0	S	0	S	S	0	0
1	0	1	S	S	S	S	U	1
S	0	S	S	S	S	S	U	S
R	0	S	S	S	S	S	U	S
F	S	S	S	S	S	S	U	S
C	0	S	S	S	S	S	U	S
U	U	U	U	U	U	U	U	U
Z	0	1	S	S	S	S	U	Z

TS BUS TRUTH TABLES

Two sets of truth tables are given for the TS BUS function. The first set are for the default DOT TS MODE. The second set are for the alternate DOR OR MODE.

TS BUS in DOT TS MODE
HARD

HARD	0	1	S	R	F	C	U	Z
0	0	U	U	C	F	C	U	0
1	U	1	U	R	C	C	U	1
S	U	U	S	C	C	C	U	S
R	C	R	C	R	C	C	U	R
F	F	C	C	C	F	C	U	F
C	C	C	C	C	C	C	U	C
U	U	U	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

TS BUS in DOT TS MODE
SOFT

HARD	0	1	S	R	F	C	U	Z
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
S	S	S	S	S	S	S	S	S
R	R	R	R	R	R	R	R	R
F	F	F	F	F	F	F	F	F
C	C	C	C	C	C	C	C	C
U	U	U	U	U	U	U	U	U
Z	O	1	S	R	F	C	U	Z

The TS BUS function for HARD/UNDRIVEN input signal strengths is identical to the HARD/SOFT table given above.

The TS BUS function for SOFT/HARD input signal strengths is obtained by transposing the values of the HARD/SOFT table given above.

The TS BUS function for SOFT/SOFT input signal strengths is identical to the HARD/HARD table given above.

The TS BUS function for SOFT/UNDRIVEN input signal strengths is identical to the HARD/SOFT table given above.

The TS BUS function for UNDRIVEN/HARD input signal strengths is obtained by transposing the values of the HARD/SOFT table given above.

The TS BUS function for UNDRIVEN/SOFT input signal strength is obtained by transposing the values of the HARD/SOFT table given above.

TS BUS in DOT TS MODE
UNDRIVEN

UNDRIVEN	0	1	S	R	F	C	U	Z
0	0	S	S	S	S	S	S	0
1	S	1	S	S	S	S	U	1
S	S	S	S	S	S	S	U	S
R	S	S	S	R	S	S	U	R
F	S	S	S	S	F	S	U	F
C	S	S	S	S	S	S	U	C
U	U	U	U	U	U	U	U	U
Z	0	1	S	R	F	C	U	Z

TS BUS in DOT OR MODE
HARD

		0	**1**	**S**	**R**	**F**	**C**	**U**	**Z**
	0	0	1	S	R	F	C	U	0
	1	S	1	S	1	C	C	1	1
	S	S	S	S	C	C	C	U	S
HARD	**R**	C	C	C	R	C	C	U	R
	F	C	C	C	C	F	C	U	F
	C	C	C	C	C	C	C	U	C
	U	U	U	U	U	U	U	U	U
	Z	0	1	S	R	F	C	U	Z

TS BUS in DOT OR MODE
SOFT

		0	**1**	**S**	**R**	**F**	**C**	**U**	**Z**
	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1
	S	S	S	S	S	S	S	S	S
HARD	**R**	R	R	R	R	R	R	R	R
	C	C	C	C	C	C	C	C	C
	F	F	F	F	F	F	F	F	F
	U	U	1	U	U	U	U	U	U
	Z	0	1	S	R	F	C	U	Z

The TS BUS function in DOT OR mode for HARD/UNDRIVEN input signal strengths is identical to

the HARD/SOFT table given above.

The TS BUS function in DOR OR mode for SOFT/HARD input signal strengths is obtained by transposing the values of the HARD/SOFT table given above.

The TS BUS function in DOT OR mode for SOFT/SOFT input signal strengths is identical to the HARD/HARD table given above.

The TS BUS function in DOT OR mode for SOFT/UNDRIVEN input signal strengths is identical to the HARD/SOFT table given above.

The TS BUS function in DOT OR mode for UNDRIVEN/HARD and UNDRIVEN/SOFT input signal strengths is obtained by transposing the values of the HARD/SOFT table given above.

TS BUS in DOT OR MODE
UNDRIVEN

UNDRIVEN	0	1	S	R	F	C	U	Z
0	0	S	S	S	S	S	U	0
1	S	1	S	S	S	S	U	1
S	S	S	S	S	S	S	U	S
R	S	S	S	S	S	S	U	S
F	S	S	S	S	S	S	U	S
C	S	S	S	S	S	S	U	S
U	U	U	U	U	U	U	U	U
Z	0	1	S	S	S	S	U	Z

SECTION 8
ERROR MESSAGES

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

The following sections define the three classes of errors, describe each error, and suggest how to correct many of them.

8.1 CLASSES OF ERRORS

Errors detected by the Timing Verifier fall into three categories:

1. Syntax Errors

   Syntax errors are typographical errors or violations in the specified form of a character string and are discovered when the system is searching any of the four input files or the design. Syntax errors are very common in signal names and properties.

   Syntax errors must be corrected before looking at timing errors.

2. Runtime Errors

   Runtime errors occur after the input files are read and while the verifier is processing the design. Runtime errors prohibit the Verifier from completing verification of the design. If the circuit does not converge, a runtime error is reported. If a required input file is lacking, a runtime error occurs.

Runtime errors must be corrected before looking at timing errors.

Timing Errors

Timing errors are design errors that cause timing problems. Common timing errors are setup and hold violations and minimum pulse width violations. If syntax and runtime errors are also present, spurious timing errors are often produced.

There are only seven different timing errors. They are numbers:

153, 156, 157, 158, 159, 160, 166

Look in the numerical listing for descriptions of each one.

8.2 FORMAT OF MESSAGES

Error documentation is included in the listing file (tvlst.dat) and a summary of errors is sent to the screen. In each case, errors are divided into the three basic types: syntax errors, runtime errors, and timing errors.

In the listing file, the format for error messages is:

#7 Syntax error(22): String length exceeded

The #7 indicates the 7th occurrence of that particular error.

The label "Syntax error (22)" tells you what type of error it is, and gives you the error number. For more complete error documentation you can look up the error in this section of the manual using the error number.

A brief message after the label describes the error. In this case it is: "string length exceeded".

Following each error message in the listing file are several lines describing the path name to the body in the drawing where the error was detected. This information helps you to locate the error on the drawing.

The summary of errors that is sent to the screen at the end of each verification run reports how many of each of the three classes of errors occurred. For example:

Twelve syntax errors detected.

No timing errors detected.

One run time error detected.

8.3 NUMERICAL LISTING OF ERROR MESSAGES

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

Each error message number has one of these four labels:

- Syntax error
- Runtime error
- Timing error
- Error # : Unused.

Unused means that the error message number is available for future use.

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

Syntax error #1: Expected identifier

This error is generated whenever the Verifier is expecting an identifier (a string of letters, digits, or '_' starting with a letter) and finds some other character. Identifiers are used

as names in properties, text macros, and as operands for Verifier directives. The Verifier prints the input line along with a pointer to the position in the line where the problem was detected.

Syntax error #2: Expected =

This error is generated whenever the Verifier is expecting an equal sign and finds some other character. Equal signs are used in many places: between property names and values, and in expressions. The Verifier prints the portion of the input line it read before it encountered the error.

Syntax error #3: Reserved.

Syntax error #4: Reserved.

Syntax error #5: Reserved.

Syntax error #6: Reserved.

Syntax error #7: Expected)

This error is generated whenever the Verifier is expecting a right parenthesis and finds some other character. The Verifier prints the portion of the input line it read before the error was encountered.

Syntax error #8: Expected ,

This error is generated whenever the Verifier is expecting a comma and finds some other character. Commas are used to separate elements in lists and are required, for example, in specifying options to the LIST command. The Verifier prints the portion of the input line it read before the error was encountered.

Syntax error #9: Reserved.

Syntax error #10: Expected <

This error is generated whenever the Verifier is expecting a < (less than character) and finds some other character. The Verifier prints the portion of the input line read before the error was encountered.

Syntax error #11: Expected >

This error is generated whenever the Verifier is expecting a > (more than character) and finds some other character. The Verifier prints the portion of the input line read before the error was encountered.

Syntax error #12: Expected ;

This error is generated whenever the Verifier is expecting a semi-colon and finds some other character. The Verifier prints the portion of the input line read before the error was encountered.

Syntax error #13: Expected :

This error is generated whenever the Verifier is expecting a colon and finds some other character. The Verifier prints the portion of the input line read before the error was encountered.

Syntax error #14: Reserved.

Syntax error #15: Expected (

This error is generated whenever the Verifier is expecting a left parenthesis and finds some other character. The Verifier prints the portion of the input line read before the error was encountered.

Syntax error #16: Reserved.

Error #17: Unused.

Error #18: Unused.

Error #19: Unused.

Syntax error #20: Unmatched closing comment character

This error is generated when the Verifier encounters a closing comment character (a right curly brace }) without a matching starting comment character (a left curly brace {). The Verifier prints the portion of the input line read before the error was encountered.

This error occurs either when the right curly brace is extraneous, or when the left curly brace was omitted. When the right curly brace really is extraneous the Verifier continues with no further errors. When it isn't, additional errors usually are generated because the Verifier tries to read the text of the comment started by a left curly brace.

Syntax error #21: Nested comments not allowed

Comments within comments are not allowed in Timing Verifier input files. This error is generated when input of the form:

{ This is a comment { This is a nested comment }}

is encountered.

Syntax error #22: String length exceeded

This error is generated as the Verifier is reading a string and finds that the string is too long. Strings are limited to 255 characters. The Verifier prints the portion of the input line read before the error was encountered. The string is

truncated at the current position and the Verifier reads until it finds the closing quote or the end of the input line. Make the string shorter.

Syntax error #23: Illegal character found

This error is generated when the Verifier finds an illegal character in an input file. All non-printing characters except TAB are illegal. The Verifier prints the portion of the input line read before the error was encountered. Remove the illegal character.

Syntax error #24: Expression value overflow

This error is generated when the Verifier evaluates an expression whose value overflows. The Verifier prints the portion of the input line read before the error was encountered. An overflow does not cause the Verifier to abort; it assigns the value 0 to the result (unless it knows a more reasonable value) and continues with the verification.

Syntax error #25: Reserved.

Error #26: Unused.

Error #27: Unused.

Error #28: Unused.

Error #29: Unused.

Syntax error #30: Reserved.

Error #31: Unused.

Syntax error #32: Non-printing character found

This error is detected when the Verifier is reading characters from an input file. A non-printing character has been. This is not permitted. The Verifier prints the portion of the input line read before the error was encountered.

Syntax error #33: Expected a string

This error is detected when the Verifier is expecting a string (a quoted sequence of printing characters) and finds something else. The Verifier prints the portion of the input line read before the error was encountered.

Syntax error #34: Comment not closed before end of input

This error is detected when the Verifier does not find the end of a comment before the end of the file. A comment is started with the left curly brace character {, and ended with the right curly brace character }. The Verifier prints the portion of the input line read before the error was encountered.

Error #35: Unused.

Syntax error #36: Reserved.

Syntax error #37: Expected.

This error is generated when the Verifier is expecting a period and finds some other character. The Verifier prints the portion of the input line read before the error was encountered. This error is most commonly caused by omitting the period following the END on the last line of the directives, case or wire delay file.

Error #38: Unused.

Syntax error #39: Reserved.

Syntax error #40: Expected END

This error is generated when the Verifier reaches what it expects to be the end of a file and no END is found. An END must be present at the end of the directives, case, and delay files. The Verifier prints the portion of the input line read before the error was encountered. The END is used to inform the Verifier that the file is complete and that it isn't unfinished or missing some text.

Syntax error #41: Reserved.

Error #42: Unused.

Error #43: Unused.

Error #44: Unused.

Error #45: Unused.

Error #46: Unused.

Error #47: Unused.

Error #48: Unused.

Syntax error #52: Reserved.

Syntax error #53: Reserved.

Syntax error #54: Reserved.

Syntax error #55: Reserved.

Error #56: Unused.

Syntax error #57: Reserved.

Syntax error #58: Reserved.

Error #59: Unused.

Syntax error #60: Reserved.

Syntax error #61: Reserved.

Error #62: Unused.

Error #63: Unused.

Error #64: Unused.

Error #65: Unused.

Error #66: Unused.

Syntax error #69: Reserved.

Syntax error #70: Reserved.

Error #71: Unused.

Syntax error #72: Reserved.

Syntax error #73: Reserved.

Syntax error #74: Reserved.

Syntax error #75: Reserved.

Syntax error #76: Reserved.

Syntax error #77: Reserved.

Syntax error #78: Reserved.

Syntax error #79: Reserved.

Syntax error #80: Output digits must be 0,1,2, or 3

This error occurs when an illegal numerical value is given to the OUTPUT_DIGITS directive. Legal values are 0 through 3. See the Directives section for more information.

Error #81: Unused.

Error #82: Unused.

Error #83: Unused.

Error #84: Unused.

Syntax error #85: Reserved.

Runtime error #86: Reserved.

Syntax error #87: Reserved.

Error #88: Unused.

Syntax error #89: Reserved.

Syntax error #90: Reserved.

Syntax error #91: Reserved.

Error #92: Unused.

Syntax error #93: Reserved.

Error #94: Unused.

Error #95: Unused.

Error #96: Unused.

Syntax error #97: Reserved.

Syntax error #98: Reserved.

Runtime error #99: Reserved.

Runtime error #100: Assertion check failure: save Log File.

This error is generated whenever the Verifier discovers some internal data problem. This message indicates an internal Verifier error and usually cannot be fixed by the user. Contact Valid Logic Systems for a work around and/or corrections. Save the data that caused the error as it will be very helpful in finding the problem. It is very important that the TVLOG file be saved (at a minimum). Valid may also request any of the input or output files for the Verifier. Try to be ready to reproduce the problem for the Service Engineer.

Runtime error #101: Cannot open compiler output (CMPEXP)

This error is generated when the Verifier is not given a ROOT_DRAWING directive and cannot find the compiler expansion file, (cmpexp.dat). If you want the Verifier to use a compiler expansion file, the file must be in the same directory where you are running the Timing Verifier. See the Compiler Reference Manual for information on using the expansion file.

Runtime error #102: Compiler expansion file has wrong type

This error occurs when an expansion file is found, but it was not produced by a compilation for time. Check that you intentionally included no ROOT_DRAWING directive and that you compiled for time. See the Compiler Reference manual for more information.

Syntax error #103: Number too large

This error is generated when the Verifier finds an integer value that is larger than 99999. This error can occur reading any of the input files: the compiler expansion file, the directives file, the case file, or the wire delay file.

Syntax error #104: Illegal character in number

This error is generated when the Verifier finds a character other than a digit or a decimal point in a number. This error can occur reading any of the input files: the compiler expansion files, the directives file, the case file, or the wire delay file.

Syntax error #105: EOF encountered

The end of an input file (EOF) was found prematurely. This means before "END." appeared in the file.

Syntax error #106: Reserved.

Runtime error #107: Reserved.

Syntax error #108: Continuation character not at EOL.

The Verifier found a line in an input file that was not ended properly. A string was being read, and it contained a newline character. Strings that extend past character 80 in an input file should include a tilda ~ (the continuation

character) to indicate that they continue on the next input line.

Syntax error #109: String too long

This error is generated when the Verifier finds a string that extends over 255 characters. Make the string shorter.

Syntax error #110: Bad delimiter

This error occurs when the Verifier is expecting some delimiter (such as a double quote ending a string), and finds a different one. The expected delimiter is printed with the portion of the input line read before the error was encountered.

Syntax error #111: Expected quoted string

This error is generated when the Verifier encounters something other than a string in quotes when it is expecting a quoted string. The portion of the input line read before the error was encountered is printed out.

Runtime error #112: Reserved.

Syntax error #113: Invalid width of signal

This error occurs when the Verifier computes a signal that has a period the is not equal to the CLOCK_PERIOD. This is a very unusual internal error, and if it ever occurs, should be reported immediately.

Syntax error #114: Reserved.

Syntax error #115: Multiple values given for signal

This error is generated when more than one value is given for a signal in one case in the case file. Check the case file for syntax problems. In particular, check that the correct lines end with commas and semicolons.

Runtime error #116: Max number of evaluation passes executed

If the circuit does not converge within the number of evaluation passes specified by the MAX_EVAL_PASSES directive (which currently defaults to 2000), this error is generated. Many evaluation passes may be required for circuits with feedback loops in them.

Runtime error #117: Resistor connected to constants at both ends

This error occurs when the Verifier tries to orient the resistors in the design. A resistor connected to constant signals (1 or 0) at both ends is not acceptable to the Timing Verifier, and usually indicates a design error.

Runtime error #118: Resistor driven at both ends

The Verifier generates this error when it finds a resistor that has primitive outputs attached to both ends. This resistor cannot be oriented. The verification run continues, but the resistor is ignored. Change the circuit so that all resistors are driven at only one end.

Runtime error #119: Part not orientable

This error is generated when a resistor cannot be oriented. Each resistor must have unique, unambiguous input and output sides; they are not allowed to be truly bidirectional.

Runtime error #120: The following parts are unorientable

This error is generated when more than one resistor cannot be oriented; see Runtime error 119 above.

Syntax error #121: Max time is smaller than min time

This error is generated by the Verifier whenever it finds a maximum time that is less than the corresponding minimum time. Specifying DELAY=5.0-4.0, for example, would cause this error.

Syntax error #122: Single time variable expected, not range

This error occurs when a minimum to maximum range (min-max) is specified where only a single time value is expected. Examples: setup times, hold times, minimum pulse widths. Check user-created timing models for errors.

Syntax error #123: Reserved.

Syntax error #124: Illegal transition type specified

This error is generated when the Verifier finds a Transition Type other than SMOOTH or GLITCHY on a primitive. Check user-created timing models for improper use of the transition property.

Syntax error #125: Illegal strength type specified

This error in generated when the Verifier finds a strength other than SOFT or HARD in a design.

Syntax error #126: Illegal character in evaluation string

Evaluation characters must be either: A, I, E, Z, H, or W. See the section on DELAYS for more information on Evaluation directives.

Syntax error #127: Bit numbers specified are out of range

This error is generated when bit subscripts specified in the case file do not agree with the signal width found in the design. If a signal has bits <5 . . 2> in the design, any specification in the case file for that signal may only refer to bits 5 through 2 or to the whole signal.

Syntax error #128: Illegal character in signal list

This error is generated by the Verifier when it finds extraneous or incorrect input in the properties connected to a signal. The various property names must be spelled correctly, and the other elements such as equal signs (=) and semi-colons (;) must be in the proper places.

Syntax error #129: Time range given for clock delay

The value of a CLOCK_DELAY property must be a single value, not a minimum - maximum range.

Syntax error #130: Undefined pin

This error is generated when an incorrect pin name is found for a Timing Verifier primitive. The correct pin names are documented in the section on Verifier primitives. Check user-created timing models.

Syntax error #131: No signal passed to parameter

This error is generated when there is no signal bound to a pin of a Timing Verifier primitive. Check that all Compiler errors have been corrected. If this error occurs when there

are no Compiler errors, it is indicative of a Compiler bug.

Syntax error #132: Missing parameter

This error is generated when a required pin of a primitive is not found in the expansion files. It occurs when optional pins on primitives, such as enables on checker primitives, are used incorrectly in library models. Check user-created timing models.

Runtime error #133: Incorrect width parameter passed to formal

This error is generated when a signal and the pin it is connected to are found to have different widths in an expansion file. Check that all Compiler errors have been corrected. This error should have caused a Compiler error to be generated.

Syntax error #134: Illegal value given to boolean option

Boolean expressions must be given either the value TRUE or the value FALSE. Any other value generates this error.

Syntax error #135: Illegal value given to on/off option

Verifier directives such as RECONV_FANOUT and DELAY_ESTIMATOR require either the value ON or the value OFF. Any other value generates this error.

Syntax error #136: Unknown dot type specified

The legal values for the DOT_TYPE directive are: DOT_OR, DOT_AND, and DOT_TS. Any other value generates this error.

Syntax error #137: Expected bit ordering specifier

This error is generated when the BIT_ORDERING directive is read, and the value assigned is not either RIGHT_TO_LEFT or LEFT_TO_RIGHT.

Syntax error #138: Too many entries given in wire estimate list

The maximum number of wire estimates that can be specified in a wire estimate list is currently 100. Any additional estimates are ignored and cause this error to be generated.

Syntax error #139: Unknown option given

This error is generated when an unknown (illegal or undefined) Timing Verifier directive is specified in the directives file. It is also generated when the value of the DELAY_MODEL directive is not one of the legal values: MIN, MAX, MIN/MAX, RISE/FALL, or a combination of the legal values. This error is most often caused by a spelling error.

Syntax error #140: Unknown syntax specification

Signal specifications have up to five parts: the property specifier, the assertion specifier, the subscript specifier, the name specifier, and the negation specifier. If the values given for the SYNTAX specification are anything else, this error is generated.

Syntax error #141: Invalid clock period specified

If the CLOCK_PERIOD is specified as less than one nanosecond, this error is generated, and the CLOCK_PERIOD set to the default, 100 nanoseconds.

Syntax error #142: Invalid number of clock intervals specified

If the number of CLOCK_INTERVALS specified in the CLOCK_INTERVALS directive is less than one or greater than 10000 times the CLOCK_PERIOD this error is generated.

Syntax error #143: Invalid tri-state bus type

The only legal values for the TS_BUS_TYPE directive are DOT_OR and DOT_TS. This error is generated when any other value is specified. The value is then set to the default, DOT_TS, or to DOT_OR if a previous TS_BUS_TYPE directive had the value DOT_OR.

Syntax error #144: NC_SIGNALS set to illegal value

The legal values for the NC_SIGNALS directive are: 0, 1, S, ASSERTED, and DEASSERTED. Using any other value causes this error and causes the value S (stable) to be used.

Syntax error #145: PULSE_EDGE_CORR must be between 0 and 1

Legal values for the PULSE_EDGE_CORR directive range between 0 and 1. See the directives summary for more information. If an illegal value is used that generated this error, the value 1 will be used as a default.

Syntax error #146: Print width invalid

Valid values for the PRINT_WIDTH directive are 80 and 132. Specifying other values generates this error. When an error occurs, the value defaults to 132. See the PRINT_WIDTH directive for more information.

Syntax error #147: Invalid number of passes specified

Specifying a value less than one for the MAX_EVAL_PASSES directive generates this error. Values less than one are meaningless, so the value defaults to 2000 when this error in encountered.

Syntax error #148: Expected FILE_TYPE

This error is generated when the Verifier finds a file called cmpexp.dat but the first characters in the file are not "FILE_TYPE". The first line of correct compiler expansion files for the Verifier is always either: "FILE_TYPE=CMP_EXPANSION;" or "FILE_TYPE=TIME_EXPANSION;". Make sure the proper expansion files exist, and that they have not been altered by hand.

Syntax error #149: Unknown primitive

This error is generated by the Verifier when it reads a primitive from the expansion file that has an unknown type. This can only happen if the expansion file was edited by hand, and a primitive's name is changed accidently. The primitive will be ignored. Do not edit the expansion files.

Syntax error #150: Expected "END_PRIMITIVE"

This is another error that is only caused by hand editing of an expansion file. Every primitive in the expansion has the keyword, "END_PRIMITIVE" at the end of its description. This error can be hard to recover from, and in some cases can cause many extraneous errors to be generated.

Syntax error #151: Unknown block type in expansion file

This error is caused by hand editing expansion files. Legal block types are: DIRECTIVES, TIME, PRIMITIVE, and END.

Runtime error #152: Reserved.

Timing error #153: Edge to Edge timing violation

This error is generated by a Timing Verifier Edge to Edge checker primitive. This indicates an edge-to edge timing error in the design. See the section on primitives for more information on this and the other Timing errors.

Error #154: Unused.

Error #155: Unused.

Timing error #156: Setup time violation

This error is generated by a Timing Verifier Setup Hold checker primitive. This indicates a setup or hold time error in the design. See the section on primitives for more information on this and the other Timing errors.

Timing error #157: Hold time violation

This error is generated by a Timing Verifier Setup Hold checker primitive. This indicates a setup or hold time error in the design. See the section on primitives for more information on this and the other Timing errors.

Timing error #158: Setup/Hold time violation

This error is generated by a Timing Verifier Setup Hold checker primitive. This indicates a setup or hold time error in the design. See the section on primitives for more information on this and the other Timing errors.

Timing error #159: Minimum pulse width timing violation

This error is generated by a Timing Verifier Minimum Pulse Width checker primitive. This indicates a minimum pulse width error in the design. See the section on primitives for more information on this and the other Timing errors.

Timing error #160: Delay is greater than CLOCK_PERIOD

Before doing the actual verification, the Timing Verifier checks that all delays are less than the CLOCK_PERIOD. Any that aren't are flagged with this error message. (This is a feature only available in release 7.5 and later releases). As always, greater delays are used modulo the CLOCK_PERIOD during the verification. That is, with CLOCK_PERIOD set to 102 ns, a delay of 104.2 ns will used as a delay of 104.2 mod 102 or 2.2ns.

Syntax error #161: Too many entries given in load coefficient table

This error is generated when more than 100 entries are given for a LOAD_COEFF table. See the section on the delay estimator under DELAYS for more information.

Error #162: Unused.

Syntax error #163: Illegal latch directive

The legal values for the LATCH_ERR_MODEL directive are: OPEN, CLOSED, and CONSERVATIVE. Specifying any other value generates this error.

Syntax error #164: Reserved (debug).

Runtime error #165: Multiple evaluation directives on primitive

The use of five of the six allowed evaluation directives is restricted to only ONE pin of a primitive. See the section on Evaluation Directives under DELAYS for more information.

Timing error #166: Input changing while clock is asserted

This error indicated that the conditions required by an A evaluation directive have not been met by the circuit. While the clock input to an AND or OR gate is asserted, the other input is not stable. See the section on Evaluation Directives under DELAYS for more information.

Syntax error #167: Reserved.

Runtime error #168: Wire-tie error - mixed and/or

This error is generated by the Verifier when some illegal combination of wire-and and wire-or logic is used on a signal. See the sections on Wire Gates and Timing Models, and the DOT_TYPE directive for more information.

Syntax error #169: Illegal value given

This error is generated when a signal is set to an illegal value in the case file. Legal values are 0, 1, S, and clock assertions. See the section on Case Analysis for more information.

Syntax error #170: Invalid casefile syntax

This error is generated when the proper case file syntax is not used. In particular, signal names and values must be enclosed in single quotes (').

Syntax error #171: Case signal not used in network

This error is generated when a signal found in the case file is not found in the design. The erroneous line in the case file is ignored.

Syntax error #172: Reserved.

Syntax error #173: Expected ; or ,

This error is generated when a line in the case file is not ended properly. Each line must end with either a comma, or a semicolon. See the Case Analysis section for the exact syntax and more information.

Syntax error #174: Signal not found - delay spec ignored

This error is generated when a signal is found in the wire delay file (delay.dat) that is not found in the design. The signal is printed out, and the specification is ignored. Usually this error is caused by a spelling error.

Syntax error #175: Signal does not drive pin, delay ignored

This error is generated when a signal is found in the wire delay file (delay.dat) with an incorrect path name. See the section on Wire Delays and Appendix A for the exact syntax required and for more information.

Runtime error #176: Dotted signal name too long

This error is generated as a warning when a signal name and its path name are too long to be concatenated to form a dotted signal name. The limit on signal name lengths is 255 characters. The Verifier picks an intelligent substitute and prints out that substitute name.

Runtime error #177: Too many outputs are wire-tied together

Currently, only 1000 primitive outputs may be wire-tied together. If this error occurs, and there is not an error in the drawing, please report the problem and the size will be increased.

Syntax error #178: Error in timing assertion

This error is generated when any one of many signal name syntax errors are detected. The exact error is printed out, and the signal in question. If possible, an intelligent substitute or default is used.

Syntax error #179: Illegal List option

This error refers to the LIST directive which has many possible options of the form <option> and NO<option>. See the LIST directive in the Directives section for more information.

Runtime error #180: No option file or TIME_DIRECTIVES block

This error is generated when the Timing Verifier cannot find any directives. The file verifier.cmd does not exist in the current directory AND there is no TIME_DIRECTIVES block in the expansion file. Create a file called verifier.cmd and enter the required directives into it.

Syntax error #181: Verification aborted-expansion file errors

This error is generated when more errors are detected while reading in the expansion file than the value given to the MAX_EXP_ERRORS directive. See the Directives section for information on the MAX_EXP_ERRORS directive.

Runtime error #182: Cannot open file for write

This error is generated by the Verifier when it cannot open the monitor (screen output) file. Check for space on the disk, no write access to the current directory, etc.

Runtime error #183: Reserved.

Runtime error #184: Verification aborted-too many input errors

This error is generated when more errors are detected while reading in the input files (not just the expansion files) than the value given to the MAX_ERRORS directive. See the Directives section for more information on the MAX_ERRORS directive.

Runtime error #185: Illegal evaluation modes

This is an internal error that very seldom occurs. If it does, please note the two evaluation modes, save the list and log files, and notify a Valid Service Engineer.

Runtime error #186: Wire table already defined, redefining

This error in generated when more than one wire estimate list is given for a single family. In the directives file or the TIME_DIRECTIVES block, more than one WIRE_ESTIMATE directive was found with either no family specification, or the same family specification.

Runtime error #187: Undefined wire delay table given

This error is generated when a FAMILY specification is given for a primitive (by attaching the body property FAMILY to the primitive), but a wire estimate list for that family is not in the directives file or in the TIME_DIRECTIVES block of the drawing.

Runtime error #188: Undefined load coefficient table given

This error is generated when a FAMILY specification is given for a primitive, but a load coefficient list for that family is not in the directives file or in the TIME_DIRECTIVES block of the drawing.

Runtime error #189: Load table already defined, redefining

This error in generated when more than one load coefficient list is given for a single family. In the directives file or the TIME_DIRECTIVES block, more than one LOAD_COEFF directive was found with either no family specification, or the same family specification.

Error #190: Unused.

Error #191: Unused.

Error #192: Unused.

Runtime error #193: No name string in print_signal_formatted

This error is an internal error that very seldom occurs. If it does, and is repeatable, please save the input and output files and report the problem to Valid.

Runtime error #194: Invalid margin in print_signal_formatted

This error is an internal error that very seldom occurs. If it does, and is repeatable, please save the input and output files and report the problem to Valid.

Error #195: Unused.

Error #196: Unused.

Error #197: Unused.

Error #198: Unused.

Error #199: Unused.

Error #200: Unused.

SECTION 9
GLOSSARY OF TERMS

Here is a glossary of important terms used in the SCALD Timing Verifier documentation. A short description of each term is given along with a reference to the section(s) where it is more fully described.

ASSERTION

A timing assertion specifies the periodic 0/1 or changing/stable behavior of a signal over time using SCALD Language syntax. The timing assertion specifies for which intervals of the clock period the signal is asserted. Timing assertions are either included as part of a signal name entered on a GED drawing, or entered in the case.dat file. Four types of timing assertions are allowed, each having one of these four prefixes: !C !P !S !D. !C and !P are clock assertions and specify 0/1 behavior. !S and !D are signal assertions and specify changing/stable behavior. See under Timing Assertions for more information.

CASE

In some circuits it is unrealistic to test for worst case timing behavior throughout the circuit. When the values of certain signals are related, and the designer knows what the few possible combinations of those signal values are, each combination of values can be specified as a case. A particular assignment of zero/one values to a set of signals in a design for a Verifier run is called a CASE. Cases are specified in the case file (case.dat). The case file is an optional input file to the Timing Verifier. See Case Analysis in Section 4 for more information.

CLOCK

A clock is a control signal whose periodic 0/1 behavior is specified with a timing assertion (either !C or !P). The timing behavior of the master clock must be specified to achieve meaningful results from the Timing Verifier.

CLOCK SKEW

Many clocks have some amount of skew over an entire system. Clock skew for the entire design can be added with two directives: CLOCK_SKEW and PREC_CLOCK_SKEW (for precision clock skew). The skew specified with the CLOCK_SKEW directive is added to all signals in the design having an !C assertion statement. The skew specified with the PREC_CLOCK_SKEW directive is added to all signals in the design having an !P assertion statement. Clock skew is always symmetrical around all clock edges. See Section 3, Directives for more information.

Asymmetrical skew can be added to any signal on an individual basis as part of a timing assertion. See under Assertion Statements for more information.

DIRECTIVE

Timing Verifier Directives are entered in the Directives File (verifier.cmd) and provide additional instructions to the Verifier. See under Directives for more information. Evaluation Directives are an exception. They are unlike all other directives and are described in Section 5, Delays. See also the definition below.

EVALUATION DIRECTIVE

An evaluation directive is a signal property attached to clock signals that instructs the Timing Verifier to treat that signal specially. Evaluation directives can be added to signal names using the test macro _E or can be added as properties to signals using the property name EVAL and a value. There are six possible evaluation directives. Five of them are used in designs that contain tuned or gated clock signals. The sixth evaluation directive _E V is used to initialize a signal to a value other than U. See under Evaluation Directives for more information.

PRECISION CLOCK SKEW

See CLOCK SKEW above.

RECONVERGENT FANOUT

When signals follow a common path, then diverge, and then reconverge the worst case delays along the two paths are related to each other. The term "reconvergent fanout" is used to describe this situation. To correctly model the delays along such signal paths and to eliminate spurious errors, the Timing Verifier compensates for the common skew of the two signals before checking for setup and hold time errors. For more information see the Directives section under RECONV_FANOUT.

SCALAR

A scalar is a discrete one-bit signal. It is not a part of a bus. All signals that are not vector signals are scalars. Typical signal names of scalars are:

ENABLE
RESET
CLOCK !C 0-4
A1
A2

In the examples above, each signal is one bit wide. A1 and A2 are unrelated signals. They are not bits of a bus A. Scalar signals never have bit subscripts because they are always only one bit wide. See also VECTOR and "Signal Name Syntax" in the SCALD Language Manual.

SIGNAL HISTORY

The Timing Verifier produces in the listing file (tvlst.dat) signal history for all signals in the design. The signal history of a signal is a list of the values of that signal (0, 1, S, R, F, C, U and Z) and the times (over a single clock cycle) when the signal has each value. An example of signal history is:

DATA S:0.0R:18.21:20.0F:31.00:32.8

If the clock period is 80ns, the signal DATA is STABLE at time 0.0, then starts to RISE at 18.2 ns, stabilizes at 1 (high) at 20.0 ns, starts to FALL at 31.0 ns, then is 0 (low) again starting at 32.8 ns. Signal history always describes cyclical behavior. So this is the behavior of the signal DATA during every clock period in which it is active. This signal illustrates wraparound. See WRAPAROUND for more information.

TEXT MACRO

Text macros are used to provide a brief way of entering information in GED (such as properties in signal names) and to allow easy global changes of frequently used values. A number of text macros have been pre-defined for particular uses. Text macros may be used in property values but not in

property names. See under Text Macros in the SCALD Language Manual for more information.

TIMING VIOLATION

There are four types of timing violations: a setup time violation, a hold time violation, a minimum pulse width violation, and an edge to edge time violation. Each of these violations is detected by a checker primitive in the timing model of the library part. An additional violation occurs when a signal's value is inconsistent with its timing assertion.

VECTOR

A vector (or vectored signal) is a signal that is one or more bits of a bus. The signal name of a vectored signal always includes a name that is common to the entire bus, and a bit subscript. The bits of a multibit signal are referred to by number. A 4-bit bus named DATA_IN has the following signal name:

DATA IN <3 . . 0>

Elsewhere on the same drawing, the signals

DATA IN <1> and DATA IN <0>

refer to two individual bits of this bus. In contrast, scalar signals are always one-bit signals. The signals

A1
A2

are two discrete signals. They are not bits of a bus named A. See also SCALAR, and "Signal Name Syntax" in the SCALD Language Manual.

WRAPAROUND

The Timing Verifier reports signal behavior in synchronous circuits. The behavior of signals is reported in terms of the CLOCK_PERIOD specified. Because the circuit is synchronous, the signal behavior of a particular signal is identical for every clock cycle during which that signal is active. Signal history does not assume that a signal is active during every clock cycle. In a counter circuit, for example, all bits of the output do not change during the same clock cycle. But in a synchronous counter, the bits that do change, change at the same time during every cycle.

For a signal DATA with this signal history:

DATA S:0.0R:18.21:20.0F:31.00:32.8

and a clock period of 80 ns, the signal history represents the behavior of DATA during the cycles in which DATA is active. Because this signal has the same value at the beginning and end of the clock period, it illustrates wraparound. If DATA is active during the first and third clock cycles, then the signal DATA is low (0) from 32.8 ns until 18.2 ns after the start of the third cycle, or 160 + 18.2 ns, and then starts to rise again. The low value (0) wraps around from the end of a period to the beginning of another.

APPENDIX A
FILE SYNTAX

A.1 CASE FILE SYNTAX

The syntax for the case file is:

<case file>::= <case list> END. |END.

<case list>::= <case>; |<case>; <case list>

<case> ::= <signal assignment list> |;

<signal assignment list>
::= <signal assignment> |
<signal assignment>, <signal assignment list>

<signal assignment>
::= <signal name> = <value> |
<signal name> = '<timing assertion>'

<value> ::= '0' |'1'

A.2 DELAY PROPERTIES SYNTAX

<delay property> ::= <property name > <value>
<value> ::= = ' <time interval specifier> '
<time interval specifier>
::= <delay range> |
<rising range> , <falling range>
<rising range> ::= <delay range>
<falling range> ::= <delay range>
<delay range> ::= <delay> |
<min delay> - <max delay>

```
<delay>        ::= <time>
<min delay>    ::= <time>
<max delay>    ::= <time>
<time>         ::= <integer> |<fixed point number>
```

A.3 WIRE DELAY FILE SYNTAX

The detailed syntax for the wire delay file is:

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

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

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

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

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

<bit number> ::= <integer>


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

<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>
```

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

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

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

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

A.4 DRIVE PROPERTY SYNTAX

<drive> ::= DRIVE = <rise and fall delay> |
DRIVE = <rise delay>,<fall delay>

<rise and fall delay> ::= <delay> |<min delay>-<max delay>

<rise delay> ::= <delay> |<min delay>-<max delay>

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

<min delay> ::= <delay>

<max delay> ::= <delay>

<delay> ::= <fixed point number>

INDEX