Report Number: WUCS-88-03

1988-02-01

LSIM2 User's Manual

Roger D. Chamberlain and Mark N. Edelman

Lsim2 is gate/switch-level digital logic simulator. It enables users to model digital circuits both at the gate and switch level and incorporates features the support investigation of the simulation task itself. Lsim2 is an augmented version of the original lsim* with the addition of several new MSI-type components models. This user's manual describes procedures for specifying a circuit in lsim2, mechanisms for controlling the simulation, and approaches to modeling systems.

LSIM2 USER'S MANUAL

Roger D. Chamberlain and Mark N. Edelman

WUCS-88-03

February 1988

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Lsim2 User's Manual

by

Roger D. Chamberlain

and

Mark N. Edelman

Lsim2 is a gate/switch-level digital logic simulator. It enables users to model digital circuits both at the gate and switch level and incorporates features that support investigation of the simulation task itself. *Lsim2* is an augmented version of the original *lsim** with the addition of several new MSI-type component models. This user's manual describes procedures for specifying a circuit in *lsim2*, mechanisms for controlling the simulation, and approaches to modeling systems.

* Chamberlain, Roger D., "Lsim: A Gate-Switch Level Logic Simulator," M.S. Thesis, Department of Computer Science, Washington University, May 1985.
Chamberlain, Roger D. and Mark A. Franklin, "Collecting Data About Logic Simulation," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-5, No. 3, pp. 405-412, July 1986.

Lsim2 User's Manual

by
Roger Chamberlain
and
Mark Edelman

There are two major tasks involved in simulating a digital circuit. The first, specifying the circuit to be simulated, is accomplished through the use of a textual circuit description language. Circuit descriptions are processed by a translator and put into a form that can be used by *lsim2*. The translation is performed by the *circ2* circuit compiler. The second task is the actual simulation of the circuit. A set of interactive commands has been included to facilitate control of the simulation from within *lsim2*. In addition, the state of the circuit can be retained by the simulator for use at a later time. This allows the user to resume work exactly where he or she left off at a previous session. The flow of information involved with the use of *circ2* and *lsim2* is represented graphically in Figure 1. This document describes the model of the real world that is implemented by the simulator, the format of the circuit description file that is input to *circ2*, and the interactive commands input to *lsim2*.

An understanding of the model implemented by the simulator is essential for the proper use of *lsim2*. There are always differences in the results of a simulation and physical reality, and an understanding of the way the simulator views the real world can help the user to minimize those differences. Also discussed are the results that can be obtained from the simulator, both about the simulated circuit and about the simulation task itself.

1. SIMULATION MODEL

An important consideration in any simulation is the model of the real world that the simulator executes. Since any model is inherently limited in what characteristics it takes into consideration, a knowledge of the way *lsim2* models digital circuits is an essential first step in the proper use of the simulator.

1.1. Logical States

The voltage levels that are associated with signal lines are modeled in *lsim2* by one of 7 logical states. These states are further divided into two major types, stable states and transient states.

There are 4 stable states:

1	high
0	low
x	undefined
z	high impedance

The high state is used to model a high voltage, or a logical "1". The low state is used to model a low voltage, or a logical "0". Undefined is used to represent an unknown state, when little is known about the voltage level of the signal. High impedance is used to model the high impedance output of components that have tri-state outputs.

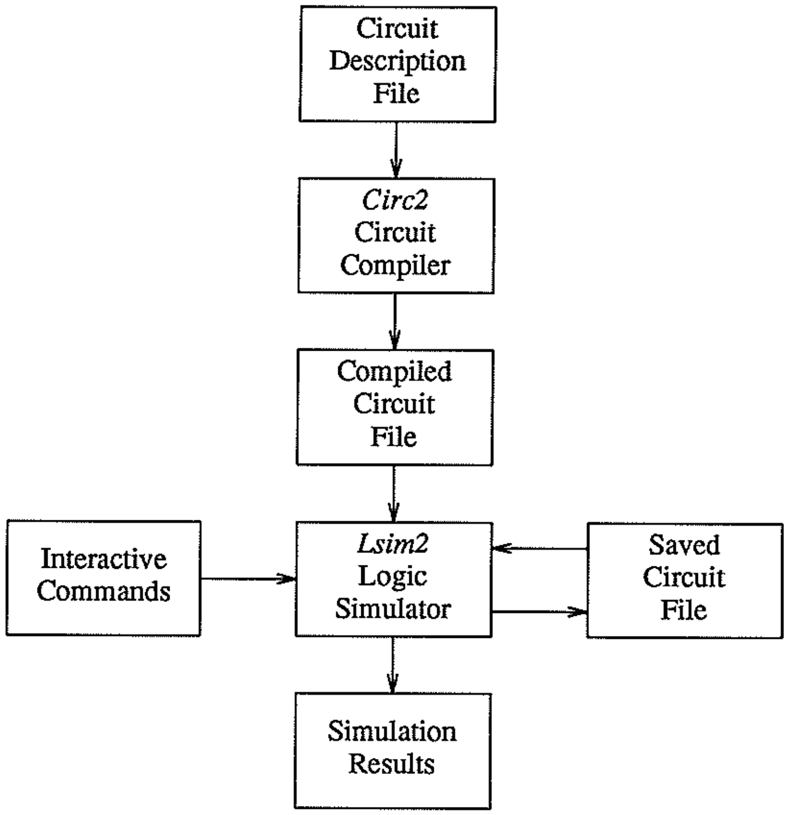


Figure 1. *Lsim2* and *Circ2*

There are 3 transient states:

r	rising
f	falling
t	transition to/from high impedance

These states are used to represent intermediate states during a transition between stable states. The rising state is used during a transition from low to high, the falling state is used during a transition from high to low, and the last state is is used as its name implies, during a transition to or from a high impedance state.

The three transient states are only utilized in the variable delay model discussed below. The unit and fixed delay models only use the first four stable states.

1.2. Current Drive Capabilities

Although *lsim2* models signal levels with 7 logical states, there are some circuit characteristics that require additional information for proper operation. The most notable characteristic is the construction known as a wired-logic OR connection, where two or more component outputs are directly tied together and drive the same signal line. If the components that are tied together have tri-state outputs and all but one of them are in the high impedance state, there is no trouble involved in determining the resulting state of the signal, it simply follows the logical state of the enabled component output. However, if two or more component outputs are enabled and are in two or more different states, additional information is needed to determine the resulting state of the signal.

For this reason, *lsim2* models the output current drive capability of a component as one of two values, either strong or weak. Strong drive capability is intended to represent a direct connection to an established voltage source, either the power supply, ground, or a connection to an active transistor whose other side is connected to power or ground. Weak drive capability is intended to represent a resistive connection to an established voltage source, such as the resistive pullup provided by a 4:1 depletion mode transistor in NMOS designs.

With the additional current drive capability information available, the simulator is now able to determine the resulting logical state of the signal in wired-logic OR connections. In a two component wired-logic OR connection, if one component has a high output and a weak high drive capability and the other component has a low output and a strong low drive capability, the resulting signal state is low. If both component outputs have the same drive capability and different output states, the resulting state of the signal is undefined.

1.3. Delay Models

There are three delay models supported within *lsim2,* the unit delay model, the fixed delay model, and the variable delay model. They each provide different types of information about the circuit being simulated.

The simplest delay model supported is the unit delay model. Timing issues are completely ignored in this model and all components are assumed to have a delay of one unit. This unit delay is not intended to have any relationship with actual time, but instead is used to provide a mechanism for providing functional simulation of the circuit without the overhead involved with more accurate timing simulations.

The second delay model supported is the fixed delay model. *Lsim2* treats each component as having a fixed low to high propagation, high to low propagation, enable, and disable time associated with each output. Whenever the component is evaluated and it is determined that an output is to change state, a component output modification event is scheduled for the current time plus the fixed time associated with the delay through the gate. For worst case analysis the maximum delay through a gate is specified as the fixed delay. However, the user can specify typical delays if he or she so desires.

The most accurate delay model takes into account the fact that not all components can have a fixed propagation delay associated with them, but are more realistically modeled by a variable time range within which the output modification is assumed to take place. The variable delay model uses a minimum and maximum value associated with each of the delay times specified, and signal levels are modeled by the transient states rising, falling, and transition to/from high impedance during the time between the known stable states.

1.4. Timing Specifications

There are a total of 12 different delay specifications associated with the delay through a component, as well as setup and hold times that can be associated with component inputs. Of the 12 delay specifications, 4 are for use with the fixed delay model and 8 are used with the variable delay model. In addition it is possible to specify multiple delays through the MSI-type components. These multiple delays allow the model to more accurately reflect the operation of more complex types of components. The timing diagram shown in Figure 2 illustrates two of the four delay specifications that are used with the fixed delay model, low to high and high to low propagation delay. The output enable time and output disable time follow a similar pattern for going to and from the high impedance state.

Timing diagrams that illustrate the variable propagation delays are given in Figure 3. The ambiguous regions are the points at which the signal is represented by one of the transient logical states. Note that the maximum time is measured from the point at which the signal reaches its final value, not from where the change begins. A general rule of thumb is that input transient states cause transient states at the component output and input stable states cause stable output states.

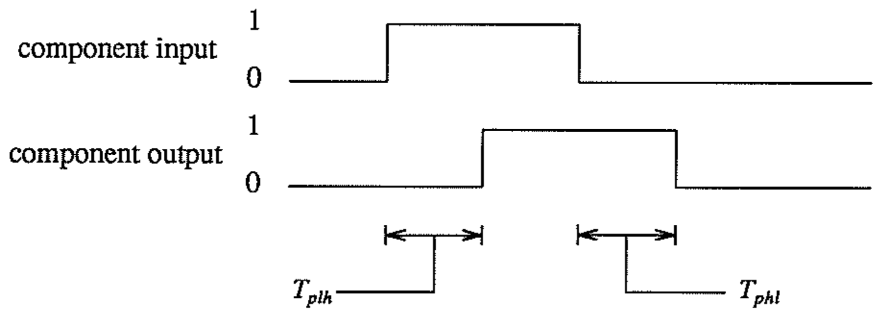


Figure 2. Fixed delay specifications

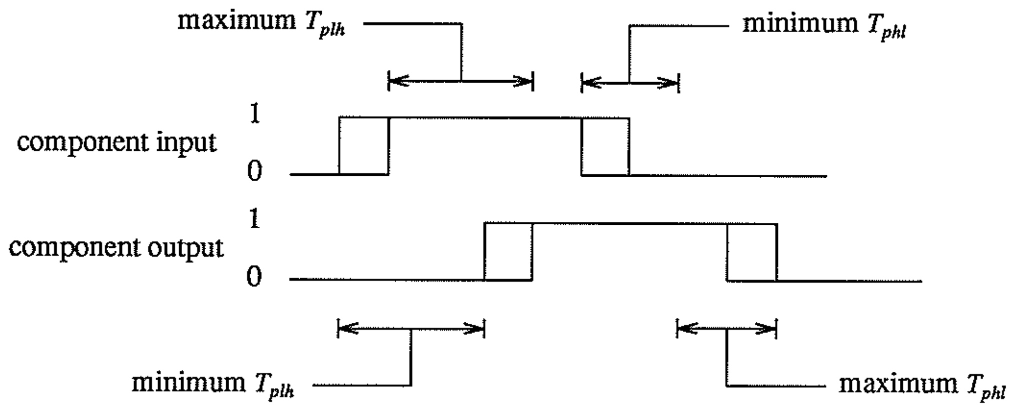


Figure 3. Variable delay specifications

Setup and hold times are used to specify input timing requirements for memory elements (flip-flops, latches, counters, etc.). The setup time associated with an input to a component tells how long the logical state of the input signal must be stable before the data is latched into the component by the clock input. Violation of the setup time is detected by the simulator and reported to the user as an error condition. The hold time tells how long the logical state of a component input must remain stable after the data has been latched by the clock input. Violation of the hold time is treated similarly to the setup time. The time the data is latched by the clock input is dependent upon the type of component involved, a level sensitive D flip flop is latched on the falling edge of the clock input, while an positive edge triggered flip flop is latched on the rising edge.

1.5. Error Detection

There are several types of errors that can occur in a digital circuit. Setup time and hold time violations have already been mentioned. In addition, *lsim2* detects spike errors and signal level errors.

A spike condition occurs when one or more input values to a component are modified before a scheduled output change has time to propagate though the component. This might be the case if there are static hazards in the circuit. Since such conditions are not necessarily considered erroneous in synchronous circuit design, *lsim2* allows error detection to be turned off using the noerr command.

A signal error results when a signal that is connected to more than one component output is driven by a strong current drive capability in more than one logical state. For example, if one component output was driving the signal with a strong ''1'' (high) state and another component output was driving it with a strong ''0'' (low) state, a signal error will result. In cases such as this, the error is reported if error reporting has not been disabled, and the logical state of the signal is set to ''x'' (undefined).

2. A SIMPLE EXAMPLE

In order to bring together an understanding of how to simulate a digital circuit with *lsim2*, a complete example is presented below.

2.1. Circuit Specification and Compilation

The first step involved in simulating a circuit is describing the properties of the circuit in a machine readable format. This circuit description must include information such as the gates to be simulated, their interconnections, delays, and other properties. In order to facilitate this description, the *circ2* circuit compiler has been developed to translate a text file description of the circuit of interest into a format readable by *lsim2*. Figure 4 is the schematic diagram of a simple three gate digital circuit that will serve as an example for explaining the use of *circ2* and *lsim2*. Note that there are labels on every component and signal line. This labeling process is the first step involved in generating a circuit description for input to *circ2*. A complete description consists of the following parts:

general delay specifications (optional)
component type definitions
macro definitions (optional)
environment specification
netlist description

This is fully discussed in later in this document. A limited discussion is given below,

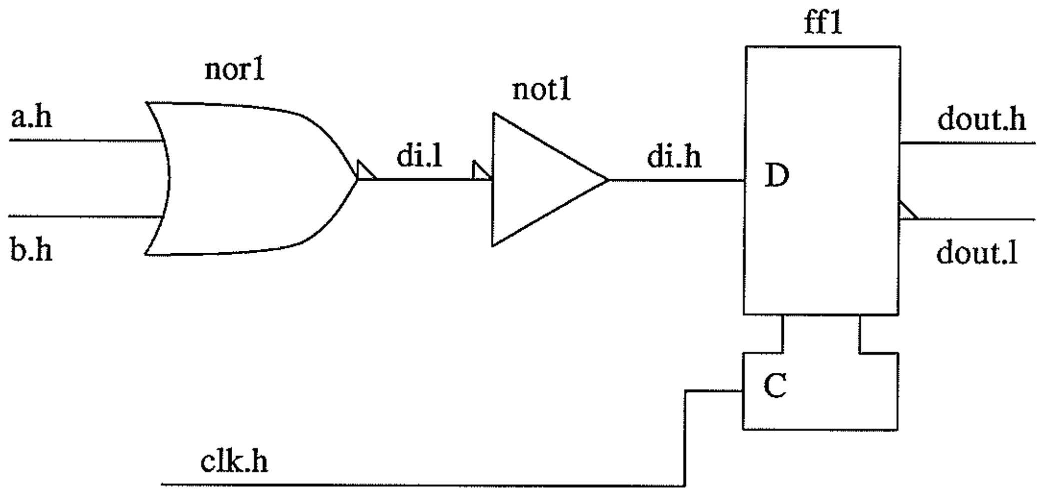


Figure 4. Example circuit

along with the description of the example circuit (Figure 5).

There are two delay specifications defined, called comdel and memdel. ''Comdel'' and ''memdel'' are user selected identifiers that will be referenced later when defining component types. They associate minimum, maximum, and fixed low to high propagation, high to low propagation, output enable, and output disable times with the given identifier.

There are three component types defined by the user in the circuit description, nmos_inv, nmos_nor, and dflip_flop. They reference the built-in functions not, nor, and dff, respectively. Not and nor are standard combinatorial gates, dff is a level sensitive D flip flop. The other parameters in the type definition indicate the number of inputs, output current drive capability, setup time, hold time, and delay specification to be associated with the type. Default values are used when a particular parameter is not explicitly given.

The environment specification defines the primary inputs to the circuit as the signals a.h, b.h, and clk.h and the primary outputs from the circuit as dout.h and dout.l.

The netlist description is where the actual components themselves first get mentioned. Nor1 and not1 are defined as components of types nmos_nor and nmos_inv respectively, with their respective input and output signals indicated. The gate named ff1 is defined as a component of type dflip_flop with input signals clk.h and di.h and output signals dout.h and dout.l. The order of the input signals is important in the description of ff1, since that is how *circ2* determines which signal is the clock signal and which is the data. The same is true for the output signals as well, the order they are specified determines which is the true output and which is the complemented output.

If the text of Figure 5 is stored in a file named circuit.ci, *circ2* must now be invoked in order to translate this text file into a format readable by *lsim2*. The following command,

```
    % circ2 circuit.ci circuit.ls
```

```
# Example circuit specification

begin circuit
    begin delays
      comdel = (8,12,12 $ 2,3,3)ns;
      memdel = (4,6,6 $ 4,6,6)ns;
    end delays;
    begin types
      nmos_inv = (not, dc=(weak, strong), comdel);
      nmos_nor = (nor, pins=(2), dc=(weak, strong), comdel);
      dflip_flop = (dff, sht=(3ns, 1ns), memdel);
    end types;
    begin environment
      inputs = (a.h,b.h,clk.h);
      outputs = (dout.h,dout.l);
    end environment;
    begin components
      nor1 = (nmos_nor, inputs = (a.h, b.h), outputs = (di.l));
      not1 = (nmos_inv, inputs = (di.l), outputs = (di.h));
      ff1 = (dflip_flop, inputs = (clk.h, di.h),
                         outputs = (dout.h, dout.l));
    end components;
end circuit;
```

Figure 5. Example circuit specification

inputs the file circuit.ci and puts the resulting translated description in the file circuit.ls. This file will be read in the next section to input the circuit description into *lsim2*.

2.2. Interactive Simulation

Once a digital circuit has been specified and compiled using *circ2*, it is ready to be simulated. If *lsim2* is invoked with circuit.ls as an argument,

```
% lsim2 circuit.ls
```

the file circuit.ls is assumed to be a previously compiled circuit description and is read in to the simulator. At this point, *lsim2* outputs a message concerning the input file and the current simulated time, outputs a prompt,

```
Simulator state input from file circuit.ls
Current simulated time = 0 units.
lsim2>
```

and waits for an interactive command to be entered by the user. Some of the more important commands are those involved with describing the inputs to the simulated circuit and the form of the output required. The following commands set up these parameters for the current example.

```
        lsim2> set 0 a.h
        lsim2> input b.h 0000000011111111 p
        lsim2> input clk.h 00001100 p
        lsim2> watch a.h b.h di.l di.h clk.h dout.h dout.l
        lsim2> output 1
```

The first command establishes a static low level (i.e. "0") at the primary input signal a.h. The second two commands define periodic waveforms to drive the primary input signals b.h and clk.h. The waveforms generated by these commands are shown in Figure 6. The b.h input has a period of 16 units and the clk.h input has a period of 8 units. The watch command tells the simulator which signals are to be output on a periodic basis, and the last command, output, specifies that the output period is to be 1 unit. In this example, the default unit delay model is used. Thus, delays through circuit components are each one generic time unit. The periods referred to in the input and output commands are also in terms of these generic units.

The status of all the signals being watched can be determined at any time during the simulation through the use of the status command. Figure 7 shows a sample terminal session that includes all of the commands that have been issued up to the current time. The "x" indicated as the logical state of the signals di.l, di.h, dout.h, and dout.l is to signify that the voltage level of the signal is undefined. The state of the other three signals was established by the previously executed set and input commands. The numbers in the final column are included to associate a signal with the appropriate column of the output of a simulation run. This will be clarified in the next couple of paragraphs.

Once the signals for the primary inputs have been established, the simulation is ready to begin. The following two commands will run the simulation for 32 time units, or 4 clock cycles with a clock period (signal clk.h) of 8 units.

```
        lsim2> cont 32
```

This command tells *lsim2* that it should continue operation until time = 32 units has been reached. The output of the simulator is given in Figure 8.

The numbers found at the top of each column of output correspond to the signals being watched. The correlation between the numbers and the signal names can be determined by examining the output of the status command given in Figure 7. The input

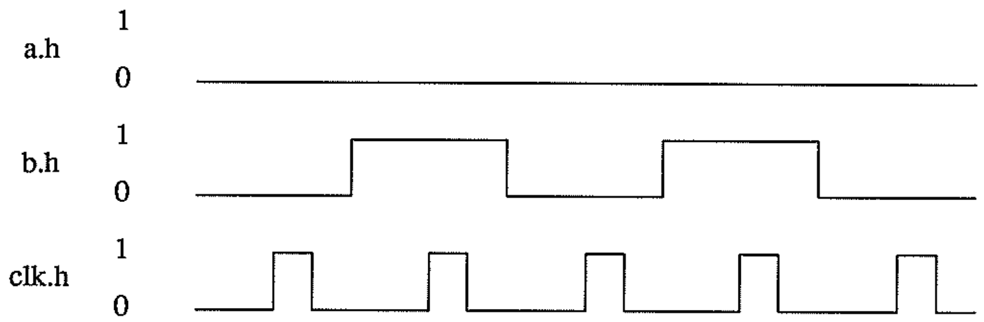


Figure 6. Input waveforms

```
% lsim2 circuit.ls
Simulator state input from file circuit.ls
Current simulated time = 0 units.
lsim2> set 0 a.h
lsim2> input b.h 0000000011111111 p
lsim2> input clk.h 00001100 p
lsim2> watch a.h b.h di.l di.h clk.h dout.h dout.l
lsim2> output 1
lsim2> status
a.h                  = 0 (watched)  (1)
b.h                  = 0 (watched)  (2)
di.l                 = x (watched)  (3)
di.h                 = x (watched)  (4)
clk.h                = 0 (watched)  (5)
dout.h               = x (watched)  (6)
dout.l               = x (watched)  (7)
lsim2>
```

Figure 7. Sample terminal session

waveforms from Figure 6 can be seen in columns 1 (a.h), 2 (b.h), and 5 (clk.h). The remaining signals can be seen to change 1 unit after their respective inputs change. Signal di.l changes 1 unit after b.h changes and di.h switches 1 unit after that. Signals dout.h and dout.l change 1 unit after the clk.h signal goes high. The vertical orientation of the output is chosen to simplify long outputs that are to be printed on continuous forms and to improve the portability of the system among standard terminals by not requiring any special curser positioning capabilities.

At this point the user could proceed in a number of directions. The current simulation could be continued by specifying another cont command, the delay model could be changed to either fixed delay or variable delay through the use of the init command, the current state of the simulation could be stored in a disk file with the save command, a previously saved simulation could be input with the read command, or the session could be terminated with the quit command. The details of all available options available for specifying circuits to *circ2* and for controlling simulations from within *lsim2* are discussed in the sections that follow.

3. CIRCUIT COMPILER

This section describes the input to *circ2,* the circuit compiler used in conjunction with *lsim2.* First a description is given of the parameters that are specified about various parts of the circuit. Then the input syntax for *circ2* is presented.

3.1. Circuit Specification

The following information is supplied to *circ2.* The input syntax used to specify the information is described in the next section.

```
lsim2> cont 32
units     1   2   3   4   5   6   7
0      |0  |0  | x | x |0  | x | x |
1      |0  |0  |  1| x |0  | x | x |
2      |0  |0  |  1|0  |0  | x | x |
3      |0  |0  |  1|0  |0  | x | x |
4      |0  |0  |  1|0  |  1| x | x |
5      |0  |0  |  1|0  |  1|0  |  1|
6      |0  |0  |  1|0  |0  |0  |  1|
7      |0  |0  |  1|0  |0  |0  |  1|
8      |0  |  1|  1|0  |0  |0  |  1|
9      |0  |  1|0  |0  |0  |0  |  1|
10     |0  |  1|0  |  1|0  |0  |  1|
11     |0  |  1|0  |  1|0  |0  |  1|
12     |0  |  1|0  |  1|  1|0  |  1|
13     |0  |  1|0  |  1|  1|  1|0  |
14     |0  |  1|0  |  1|0  |  1|0  |
15     |0  |  1|0  |  1|0  |  1|0  |
16     |0  |0  |0  |  1|0  |  1|0  |
17     |0  |0  |  1|  1|0  |  1|0  |
18     |0  |0  |  1|0  |0  |  1|0  |
19     |0  |0  |  1|0  |0  |  1|0  |
20     |0  |0  |  1|0  |  1|  1|0  |
21     |0  |0  |  1|0  |  1|0  |  1|
22     |0  |0  |  1|0  |0  |0  |  1|
23     |0  |0  |  1|0  |0  |0  |  1|
24     |0  |  1|  1|0  |0  |0  |  1|
25     |0  |  1|0  |0  |0  |0  |  1|
26     |0  |  1|0  |  1|0  |0  |  1|
27     |0  |  1|0  |  1|0  |0  |  1|
28     |0  |  1|0  |  1|  1|0  |  1|
29     |0  |  1|0  |  1|  1|  1|0  |
30     |0  |  1|0  |  1|0  |  1|0  |
31     |0  |  1|0  |  1|0  |  1|0  |
Simulation halted at time = 32 units.
lsim2>
```

Figure 8. Simulator output

Signals

Each signal is given a unique name for identification.

Components

> unique name for identification
> logic function
> number of inputs and outputs (only for variable input or output gates)
> names of input signals
> names of output signals
> output signal driving capability
> parameters for tailoring operation of MSI components
> timing information
>> low to high, high to low propagation times
>> output enable time, output disable times
>> setup, hold times

Some of the above information has default values associated with the logic function, but these may be overridden for individual components. The logic functions that are available for defining circuits are described in Appendix A. Along with the functions themselves, the appendix gives the default values for the other parameters associated with the function.

3.2. Macro Definitions

Macro definitions are allowed in the circuit description language. The macro circuit is described just as any other circuit, with the exception that the macro must be given a name so that it can later be specified as a component in a larger circuit. When referencing a macro, the macro name is given as the logic function of the component in place of one of the built-in functions. Macros may be nested, as long as the referenced macro has been defined earlier. Recursive macro definitions are not supported.

The macro facility is provided for two reasons. One, if the circuit consists of multiple copies of a group of components, macro definitions can greatly decrease the quantity of user generated input required to specify a circuit. This is often the case for digital circuits. Two, the ability to define a large functional unit as a macro can help simplify the specification of a large circuit by allowing one to include an independently specified subsection of the circuit without concern for naming conflicts between the subsection and the remainder of the circuit. This is analogous to the ability to use the same name for more than one variable in block structured programming languages.

3.3. Input Syntax

The following is a description of the input syntax of *circ2*, the circuit compiler designed for use with *lsim2*. The circuit description is divided into five sections, delay specifications, type definitions, macro definitions, environment specification, and netlist description. All identifiers in the circuit description must consist of a sequence of letters, digits, and the characters '.', '_', '[', ']', and '´', starting with a letter or the character '['. Comments begin with a '#' and continue to the end of the line. The overall block structure is as follows:

```
begin circuit
        delay specifications
        type definitions
        macro definitions
        environment specification
        netlist description
end circuit;
```

The delay specifications and macro definitions are optional.

The first section, delay specifications, associates a name with a set of delay values. The name is later referenced when the delay values are to be associated with component types and individual components. The syntax is:

```
begin delays
        delayname = (low to high propagation times $
                    high to low propagation times $
                    enable times $ disable times) timescale;
                .
                .
                .
        delayname = (low to high propagation times $
                    high to low propagation times $
                    enable times $ disable times) timescale;
end delays;
```

where each of the four times may consist of:

minimum time, maximum time, fixed time

for the corresponding change. Not all four times need be specified, default conditions are interpreted as follows:

(a)	→	(a $ a $ 0,0,0 $ 0,0,0)
(a $ b)	→	(a $ b $ 0,0,0 $ 0,0,0)
(a $ $ c)	→	(a $ a $ c $ c)
(a $ b $ c)	→	(a $ b $ c $ c)

The meaning of each time specified is somewhat self-evident, the low to high propagation times refer to the time required for a low to high transition of a component output, the high to low propagation times refer to a high to low transition, the output enable times refer to the time required to leave a high impedance state, and the output disable times refer to the time required to enter a high impedance state. The time scale is one of the following:

ms	milliseconds
us	microseconds
ns	nanoseconds
ps	picoseconds

The default scale is nanoseconds. The following conditions must hold for the delay times:

minimum time $\leq$ fixed time $\leq$ maximum time

If the fixed time is not given, it defaults to the maximum time. If either the minimum or maximum times are not given, they default to the fixed time. If only one time is given, all three times take the given value. The above rules apply to all time specifications, low to high and high to low propagation times, enable times, and disable times, and are checked by *circ2* to ensure that they are met.

The following delay specification is from the example circuit described previously:

```
begin delays
     comdel = (8,12,12 $ 2,3,3)ns;
     memdel = (4,6,6 $ 4,6,6)ns;
end delays;
```

It defines two delay specifications, called comdel and memdel. Comdel indicates a minimum low to high propagation delay of 8 ns, a maximum and fixed low to high propagation delay of 12 ns, a minimum high to low propagation delay of 2 ns, and a maximum and fixed high to low propagation delay of 3 ns. The enable and disable times all default to 0 ns. Memdel has a minimum low to high and high to low propagation delay of 4 ns, as well as maximum and fixed propagation delays of 6 ns. Its enable and disable times also default to 0 ns.

The delay specifications given above provide sufficient information for the simulator to model the circuit using a unit, fixed, or variable delay model. The model that is actually used depends upon the interactive commands given by the user during the simulation run.

With the MSI-type components it is possible to define more than one delay for use in the operation of a single component. These multiple delays are named using a base name and a set of input/output suffix labels. The details of this naming scheme will be covered in the coming sections.

The second section of the input to *circ2*, type definitions, associates the following information about a component type with a unique name to identify the type:

logic function
number of inputs and outputs
setup and hold times
output driving capability
parameters for tailoring operation of MSI components
delay specification

The syntax of the type definition section is:

begin types
typename = (function, **pins**=(inputs,outputs),
sht=(setuptime, holdtime),
dc=(high_drive_capability, low_drive_capability),
param=(**md**=mode,**wid**=width,**sz**=size,
msc=miscellaneous,**clkopt**=clock_options,
clropt=clear_options)
delayname);
.
.
.
typename = (function, **pins**=(inputs,outputs),
sht=(setuptime, holdtime),
dc=(high_drive_capability, low_drive_capability),
param=(**md**=mode,**wid**=width,**sz**=size,
msc=miscellaneous,**clkopt**=clock_options,
clropt=clear_options)
delayname);
end types;

The parameters associated with a type definition are listed below:

function	reference to a built-in function
pins	ordered pair of number of inputs and number of outputs
sht	ordered pair of setup time and hold times
dc	output high and low current drive capability
param	list of MSI specific parameters
delayname	previously defined delay specification

The only essential piece of information is the function (and the pins and param sections for the MSI components). All other parameters have a default value associated with the function. These default values, along with a list of the available built-in functions, are provided in Appendix A.

Again referring to the example described previously, the following type definitions were made:

```
begin types
        nmos_inv = (not, dc=(weak, strong), comdel);
        nmos_nor = (nor, pins=(2), dc=(weak, strong), comdel);
        dflip_flop = (dff, sht=(3ns, 1ns), memdel);
end types;
```

There are three type definitions given in the above example. Nmos_inv references the built-in function not, specifies a weak, or resistive, high drive capability, specifies a strong low drive capability, and references the previously defined comdel delay specification. Nmos_nor references the nor built-in function, indicates that there are to be two input signals (*pins = (2)*) to components of this type, and specifies the same drive capability and delay specification as the nmos_inv type. The dflip_flop type definition references the built-in function dff, a level sensitive D flip flop that retains the D input at its outputs when the clock input goes low, provided the setup and hold times are met. It has a setup time of 3 ns, a hold time of 1 ns, and references the memdel delay specification. The drive capability for this type defaults to strong high and strong low.

In addition to the delay explicitly associated with the component type through the type declaration, the MSI components can utilize additional delays to allow different delays to be associated with different input/output pairs. These delays are referenced through the named delay making use of several two letter suffix combinations. The delay name given in the type specification is used as a base for the determination of the alternative delays to use. The alternative names are formed by replacing the last two characters of the original delay name with the two letter suffix that corresponds to the correct input/output pair. If a delay with the indicated name is not found, the origional delay remains in use. For example, an MSI RAM component has four delay suffixes:

eo - output enable delay
do - data in to data out delay
ao - address in to data out delay
co - chip select in to data out delay

If the delay declaration section included the following:

```
        ram1eo = (12,20,15);
        ram1do = (25,50,30);
        ram1ao = (17,45,20);
```

and the type declaration for this RAM was:

```
ram_1 = (ram,pins=(8,3),param=(md=0,wid=3,sz=3,msc=100),ramleo);
```

then output enable signal changes would utilize the delays specified by ram1eo, data input changes would utilize those specified by ram1do, and address input changes those specified by ram1ao. Since the fourth suffix combination, co, was not defined, the delays used when the chip select input changes will be those specified by the delay explicitly associated with the type declaration (ram1eo in this case).

The third section consists of macro definitions. Macros are used to associate a single name with a group of components. The syntax for defining macros is as follows:

```
begin macro macroname
        environment specification
        netlist description
end macro;
    .
    .
    .
begin macro macroname
        environment specification
        netlist description
end macro;
```

The environment specification and netlist description are syntactically the same as the sections described below. The environment specification defines the inputs and outputs of the macro, and the netlist description defines the internal details of the macro.

The fourth section of the input to *circ2* is the environment specification. This section identifies the primary inputs and outputs of a circuit or a macro definition. The input syntax is as follows:

```
begin environment
        inputs = (signalname, signalname, ... );
        outputs = (signalname, signalname, ... );
end environment;
```

Circ2 requires that all signals be driven by at least one component output or be listed as primary inputs. An error message is generated if this requirement is not met. Although it is possible to set the value of any signal, periodic input waveforms can only be specified for primary inputs.

The final section is the netlist description. This is where the individual components that make up the circuit or macro are specified and their interconnections indicated. The type definitions given previously are referenced here and the following additional information is given for each component:

unique name
names of component input signals
names of component output signals
mask information (for ROM and PLA components only)

The syntax for the netlist description is:

```
begin components
          componentname = (typename,
                              inputs = (signalname, signalname, ... ),
                              outputs = (signalname, signalname, ... ),
                              mask = (mask_type,mask_contents));
                    .
                    .
                    .
          componentname = (typename,
                              inputs = (signalname, signalname, ... ),
                              outputs = (signalname, signalname, ... ),
                              mask = (mask_type,mask_contents));
end components;
```

The syntax for the mask portion of the netlist description is:

mask = (**ROM,** word1,word2,word3,...);

for a ROM component and:

mask = (**PLA,** eqn1,eqn2,eqn3,...);

for a PLA component, where word1, word2, word3, ... are decimal values that represent the contents of the ROM being specified. If the width of the binary equivalent for the decimal value given for a word is smaller than the word width of the ROM, the given value is padded with zeros to match the ROM word length. Similarly, if the required width is greater than the ROM word width the excess information is truncated.

The syntax for the PLA equations is:

output_name = product + product + ...

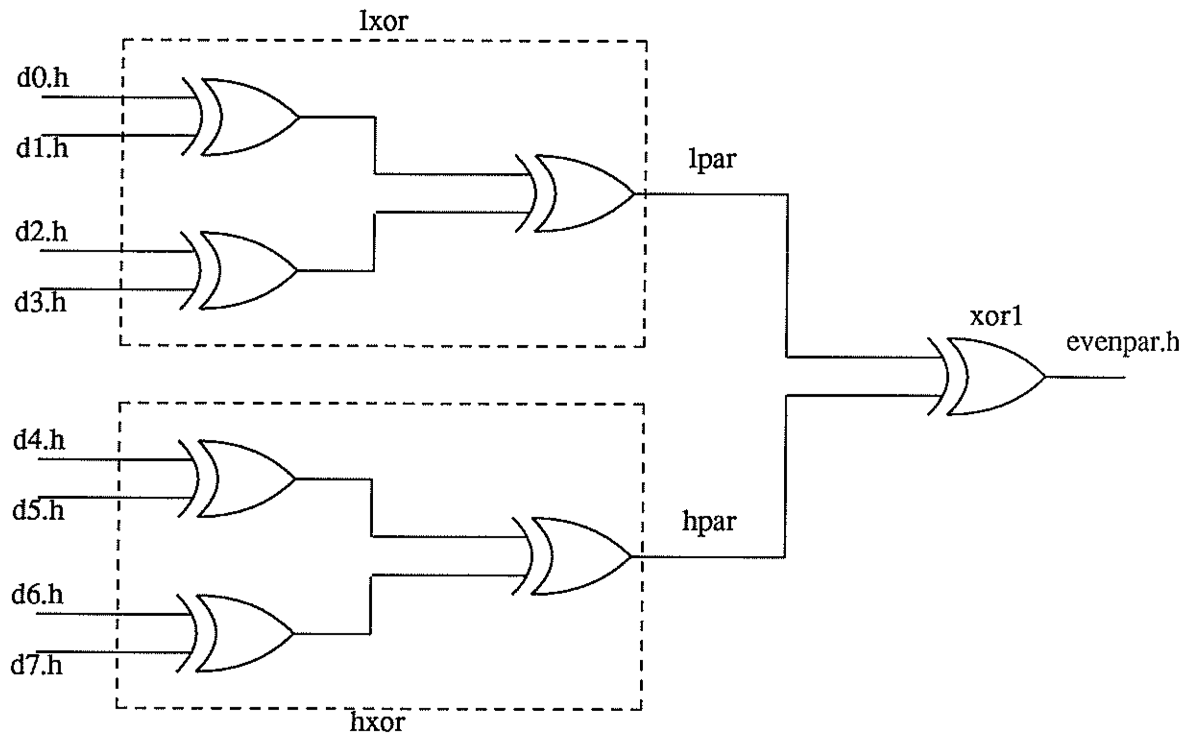
where + stands for a logical OR operation. A product is defined as:

product = input_name * input_name * ...

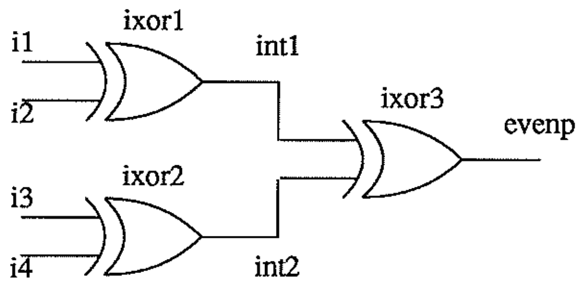
with * standing for a logical AND operation. The names used within the equation descriptions should correspond to the names given in the first portion of the netlist description for the PLA component. The input_name may be prefaced with an exclamation mark (!) to denote complementation of the input signal within the product term.

The typename can be either a type defined in the type definitions section of the circuit description or a macroname defined in the macro definitions section. If it is a macro, the input signals and output signals refer to the primary inputs and outputs indicated in the macro definition. Additionally, if the typename refers to a macro, the mask option is not valid since that option is utilized only at the component level. It is important for the number of inputs and outputs to correspond exactly with the number in the referenced type or macro, otherwise an error will result. If the netlist description is within a macro definition, other macros can still be referenced as component types, provided that the referenced macro has been previously defined. In other words, no forward references are allowed when defining macros.

The following example illustrates the use of macros in describing a circuit for input to *circ2*. The circuit to be described, shown in Figure 9(a), is an 8 bit even parity generator. The two blocks of the circuit that are surrounded by the dotted lines are the same, and are therefore excellent candidates to be defined as a macro. The circuit inside the blocks is shown again in Figure 9(b). The macro is given the name ''evenpar4'' to signify that it is a 4 bit even parity generator. The complete circuit description is given in



(a) complete circuit



(b) macro evenpar4

Figure 9. Eight bit parity generator

Figure 10. The components lxor and hxor are actually references to the macro evenpar4, and are expanded into the 3 components that make up the macro during compilation by *circ2*. The inputs and outputs listed with lxor and hxor are placed into one to one

```
# Eight bit parity generator

begin circuit
    begin delays
        xordel = (10 $ 10)ns;
    end delays;
    begin types
      ttl_xor = (xor, dc=(strong, strong), xordel);
    end types;
    begin macro evenpar4
        begin environment
            inputs = (i1,i2,i3,i4);
            outputs = (evenp);
        end environment;
        begin components
            ixor1 = (ttl_xor, inputs=(i1,i2), outputs=(int1));
            ixor2 = (ttl_xor, inputs=(i3,i4), outputs=(int2));
            ixor3 = (ttl_xor, inputs=(int1,int2), outputs=(evenp));
        end components;
    end macro;
    begin environment
        inputs = (d0.h,d1.h,d2.h,d3.h,d4.h,d5.h,d6.h,d7.h);
        outputs = (evenpar.h);
    end environment;
    begin components
        lxor = (evenpar4, inputs=(d0.h,d1.h,d2.h,d3.h),
            outputs=(lpar));
        hxor = (evenpar4, inputs=(d4.h,d5.h,d6.h,d7.h),
            outputs=(hpar));
        xor1 = (ttl_xor, inputs=(lpar,hpar), outputs=(evenpar.h));
    end components;
end circuit;
```

Figure 10. Parity generator circuit description

correspondence with the signals listed as inputs and outputs in the environment specification within the macro definition.

There are a couple of things worth noting at this point that pertain to the use of macros. Note that the components and signal lines inside the macros are not labeled in Figure 9(a). When the user wishes to identify a particular point within a macro to the simulator, the name is input by identifying the macro, typing a '/', and then identifying the point within the macro. For example, the command

```
watch lxor/int1
```

tells *lsim2* to watch the int1 signal within the lxor macro. Unique points within nested macros are identified by extending the above notation with additional macro names separated by '/' characters. The signals listed in the environment specification of the macro are referenced by the name under which they are known in the circuit description one level up, where the macro is called and input and output signal names are given.

4. INTERACTIVE CONTROL

The parameters used in controlling the simulated circuit fall into two major categories, those that are fixed for the duration of a simulation run, and those that can be modified during the run. Fixed parameters include the number of logical states used in the simulation, the delay model used in the simulation, and the resolution of simulated time (the smallest increment of time representable during the simulation). Modifying these parameters requires the circuit to be reinitialized and the simulation to start again at time = 0. The second category is the set of dynamically modifiable parameters. These parameters may change several times during a single simulation run. They generally control the operation of the simulator: input specification, output specification, traced signals, watched signals, error reporting, forced signals and components, and data collection.

4.1. Fixed parameters

Parameters that are fixed for the duration of a simulation run:

Number of logical states

4	high, low, high impedance, undefined (for use with unit or fixed delay model)
7	high, low, high impedance, rising, falling, transition to/from high impedance, undefined (for use with variable delay model)

Delay model
- unit delay
- fixed delay
- variable delay

Time resolution

The unit delay model is used to test the functional accuracy of the circuit, without concern for timing issues. All components are simulated with a delay of 1 unit, where the unit is not associated with any real time value. This model provides for the fastest execution time, but provides only functional information about circuit performance.

The fixed delay model is used when a more accurate timing model is desired. Components are modeled by a fixed time delay, for worst case analysis a maximum propagation delay would be used. The delay can vary from component to component, and the low to high and high to low propagation delays can be different as well.

The most accurate timing model supported is the variable delay model. Propagation delays through individual components are characterized by a maximum and a minimum value, and the output signal goes to an intermediate logical state during the time interval between the maximum and minimum propagation delays.

There is a potential tradeoff between the execution time of the simulator and the resolution of the simulated time clock. This is due to the assumptions made about event distribution that were used to speed up the event queue processing. With a very fine simulated time resolution it is possible to defeat the assumptions, slowing down the processing of the event queue and thereby slowing down the simulation as a whole. As long as the ratio of the time resolution to the maximum delay specification is somewhat less than 1000, the assumptions should not be endangered and no performance degradation should result. For example, if the time resolution is specified at .1 ns, there should be no problem with the above considerations as long as the maximum delay specification for components in the circuit is less than 100 ns.

4.2. Simulation startup

In order to startup *lsim2* and initialize the simulation, the following command is given to the operating system,

```
% lsim2 circuit.ls circuit.init
```

Circuit.ls is assumed to be the output of the *circ2* circuit compiler or the result of a previously executed save command from within *lsim2*. *Lsim2* will initially execute interactive commands from the file circuit.init if present, as well as from the file .lsimrc in the user's home directory.

4.3. Interactive commands

There are a host of commands that can be interactively entered during the simulation. These commands are provided to control the runtime operation of *lsim2*. They generally provide control over the set of dynamically modifiable parameters associated with the simulation, including input specification, output specification, tracing signals, watching signals, error reporting, forcing stuck-at conditions on signals and components, and controlling data collection. They also provide control over the fixed, or static, parameters described previously. Commands are normally terminated with newline, but can carry over to multiple lines by preceding the newline with a '\'. When specifying a list of components or signals as arguments to a command, the standard wildcard characters, '*' and '?', operate as one would expect. All of the commands can be given through the use of an indirect command file as well as directly from the terminal. The commands are as follows:

1. Alias command

alias
alias id
alias id command

Associate the string id with command so that id can subsequently be used in place of command as an interactive input. When no arguments are specified, the current list of aliases is reported. If only id is given, the alias associated with id is reported. For example,

```
lsim2> alias s set 0
lsim2> s a.h
```

will set the signal a.h to 0.

2. Collect command

collect on
collect on file
collect out
collect out **-p**
collect out **-c** component ...
collect off
collect off **-p**
collect off **-c** component ...

Turn the data collection facilities on or off, or output data collection results, depending on the argument given. Data collection is initially off. This is useful for monitoring small parts of the simulation without clouding the data with uninteresting portions of the task, such as reading in the circuit description from a disk file. If file is specified, raw data is output to the file suitable for input to the *S* statistical analysis package. The ''-p'' flag causes communication between partitions to be reported. The ''-c'' flag causes the reporting of the number of events processed for each component listed. The two flags can be combined in one command.

3. Cont command

cont
cont time

Continue the simulation for the specified time. If time is not specified, the time given with the last invocation of the cont command is assumed. This command allows the user to easily run the simulation for a specific amount of time in a repetitive fashion, as is often the case when simulating clocked circuits.

4. Debug command

debug on
debug off

Turn the debugging output on or off, depending on the argument given. The initial value is off. The output supplied includes messages concerning events being scheduled, retrieval of events from the event queue, and a variety of other messages that are conditional on compile time switches.

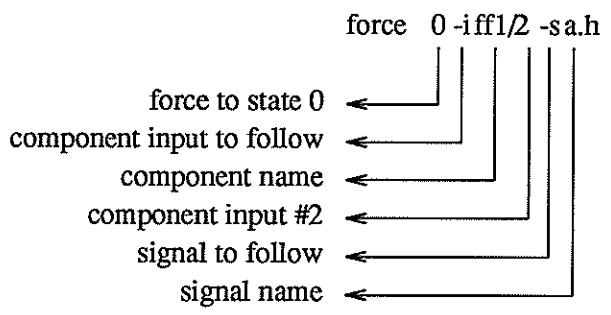
5. Force command

force state signal ...
force state **-i** component/inp ...
force state **-o** component/outp ...

Force the level of the specified signals, component inputs, or component outputs to the given logical state. This simulates a stuck-at condition for fault simulation. For example,

```
lsim2> force 1 a.h
```

puts the a.h signal in a stuck-at-1 state. The logical state of this signal now cannot be modified by the simulator until it is explicitly freed using the free command described below. Component inputs and outputs are numbered starting with 1, so the output of a gate with only a single output would be specified by component/1. The ''-i'' and ''-o'' signify that the identifiers to follow are component inputs and outputs, respectively. They do not have to be immediately after the state specification, but may follow a list of signals. A ''–s'' option is also available, to signify that the identifiers to follow are signals. This is to allow signals to follow component inputs and outputs on the command line. For example,

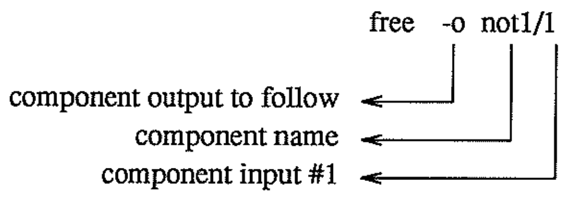


would put both the second input to the component ff1 and the signal a.h in a stuck-at-0 state. Wildcarding is not allowed for component inputs and outputs, but is supported for signals.

6. Free command

> **free** -**i** *
> **free** -**o** *
> **free** signal ...
> **free** -**i** component/inp ...
> **free** -**o** component/outp ...

Free the specified signals, component inputs, or component outputs from their stuck-at fault condition as established by the force command. This allows the simulator to set their logical state. The "-i", "-o", and "-s" options work as in the force command. For example,



frees the first output of component not1 from a previously specified stuck-at condition. Specifying "*" as the argument frees all the component inputs or outputs that are currently being forced. This is the only wildcarding allowed for component inputs and outputs. Signal wildcarding is fully supported.

7. Halt command

> **halt** time

Halt the simulation at the specified simulated time. This command allows the user to regain control of the simulation at some predetermined simulated time.

8. Init command

> **init**
> **init** delmodel
> **init** time
> **init** delmodel time

Reinitialize the simulator state, setting all signals to "x" (undefined) except the primary inputs. Delmodel must be one of "unit", "fixed", or "variable". If specified, it is used as the delay model in further simulation. The initial delay model is unit delay. Time is used to set the resolution of the simulated time clock, the initial value of which is 1 ns. This is the command that sets the fixed parameters described in the previous section. Note that these parameters can only be changed when the simulator state is being reinitialized. For example,

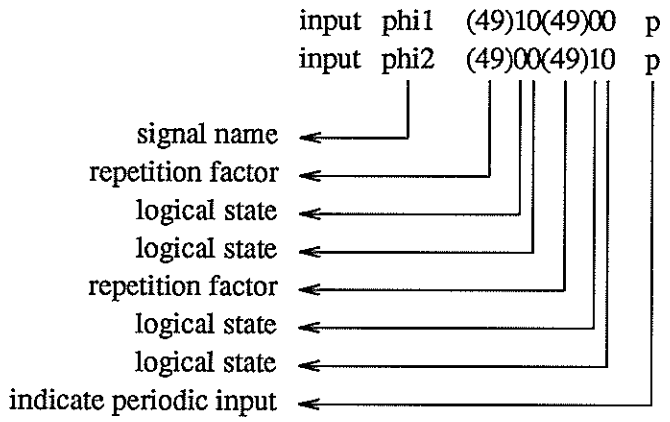
```
        lsim2> init variable 100 ps
```

reinitializes the simulator, indicates the variable delay model is to be used, and specifies the resolution of the simulated clock as 100 ps, or .1 ns.

9. Input command

> **input** signal statelist
> **input** signal statelist **p**
> **input** signal statelist time
> **input** signal statelist **p** time

Specify periodic input for signal. Statelist is the sequence of logical states for the signal to traverse. A repetition factor may precede a state in the list if it is delimited by parentheses. If the "p" is given, the sequence of states is assumed to be repeating. This is the mechanism used to specify periodic input waveforms, such as clock signals. Time is the time associated with each state in the list, the default value is 1 ns. For example,



```
input  phi1  (49)10(49)00  p
input  phi2  (49)00(49)10  p
```

Defines two periodic inputs on the signals phi1 and phi2, each with a period of 100 ns. The signal phi1 is high for 49 ns and then low for 51 ns. The signal phi2 is low for 50 ns, high for 49 ns, and then low for 1 ns. These two signals could then be used as a 10 MHz two phase non-overlapping clock input. The signal must be a primary input, one of the signals given in the environment specification of *circ2*.

10. Link command

> **link** file
> **link** file entrypoint ...

Dynamically link the specified file to *lsim2* so that the entrypoints can be called using the run command. If no entrypoints are specified, "_sim" is assumed.

11. Noerr command

> **noerr** signal ...
> **noerr -c** component ...

Ignore error messages concerning the given signals or components. The "-c" signifies that the identifiers to follow are components. It does not have to be immediately after the noerr command, but may follow a list of signals. A "-s" option is also available, to signify that the identifiers to follow are signals. This is to allow signals to follow components on the command line. For example,

```
        lsim2> noerr -c *
```

turns off error reporting for all components in the circuit; spike errors, setup time violations, and hold time violations are not longer reported to the user when they occur.

12. Output command

> **output** time
> **output -t** time
> **output** off

Periodically output the logical state of watched signals. The period is set by the input time. If the "-t" option is given, output is assumed to be going to the terminal and the column headers are repeated every 24 lines. If the argument is "off", periodic output is stopped.

13. Quiet command

> **quiet** on
> **quiet** off

Turn quiet mode on or off, depending on the argument given. Quiet mode determines whether the commands executed as the result of a source command or the third argument to *lsim2* are echoed to the terminal. The initial value is off, echoing takes place. This command is useful when using long indirect command files and it is annoying to watch all the output that is generated.

14. Quit command

> **quit**

Exit *lsim2* and return to the operating system.

15. Read command

> **read** file

Input a circuit description from file. The file should either be the output of the *circ2* circuit compiler, or the result of a previously executed save command.

16. Repterr command

> **repterr** signal ...
> **repterr -c** component ...

Report error messages concerning the given signals or components. This is the default condition for every circuit location. The repterr command is provided to negate the affects of a previously specified noerr command. The "-c" and "-s" options work as in the noerr command.

17. Run command

> **run**
> **run** entrypoint

Execute the code at the specified entrypoint. If the entrypoint is not given, "_sim" is used. It is assumed that the entrypoint was previously linked to *lsim2* using the link command.

18. Save command

> **save** file

Retain the current state of the simulator in file. This file can later be input with the read command to continue the simulation at the present point.

19. Set command

> **set** state signal ...
> **set** state **-c** component/outp ...

Set the logical state of the specified signals or component outputs to the given input state. The signals are only set once and can be overridden at a later time by the simulator if the signal specified is driven by one or more component outputs. The component outputs can be overridden if the inputs to the component change state. Signals do not have to be primary inputs as in the input command. For example,

```
        lsim2> set x ena.h d1.h
```

sets the signals ena.h and d1.h to the x (undefined) state. If either of the two signals are connected to the output of a component and the component changes state, the signal is not forced to the undefined state, but instead will follow the component output. Component outputs are numbered starting with 1, so the output of a gate with only a single output would be specified by component/1. The "-c" signifies that the identifiers to follow are component outputs. It does not have to be immediately after the state specification, but may follow a list of signals. A "-s" option is also available, to signify that the identifiers to follow are signals. This is to allow signals to follow component outputs on the command line. Wildcarding is not allowed for component outputs, but is supported for signals.

20. Setram command

> **setram** comp addr data ...

Set the internal state of a RAM component. The internal state of *component* starting at address *addr* is set to the value *data.* Additional data values are stored at subsequent addresses in the RAM. Addresses and data values can be supplied in decimal or in hex. Hex is indicated by prepending the number with a 0x (e.g., 0x1c = 28).

21. Sh command

> **sh**

Invoke an interactive version of the shell. If the environment variable SHELL cannot be found, */bin/csh* is invoked. This command is useful only in the UNIX environment, and a compile time switch is included to enable or disable the execution of sh.

22. Show command

> **show** signal ...
> **show -c** component ...

Output the logical state of the specified signals and/or components. The "-c" signifies that the identifiers to follow are components. It does not have to be immediately after the show command, but may follow a list of signals. A "-s" option is also available, to signify that the identifiers to follow are signals. This is to allow signals to follow components on the command line.

23. Showram command

> **showram** component
> **showram** component addr
> **showram** component addr addr

Query the contents of a RAM component. The internal state of *component* is displayed. If *addr* is not supplied, the entire contents of the RAM are shown. If one *addr* is given, the data at that address is shown, and if both *addr's* are given, they are taken to be an address range and data between the two addresses is shown. Any bit values not "0" or "1" cause the entire word to be shown as undefined.

24. Source command

source file

Execute interactive commands from file. If the quit command is not present in the indirect command file, return to interactive input on completion. The source command can be nested.

25. Start command

start

Initiate the simulation. The simulation stops when the simulated time specified in a halt command is encountered, or the event queue becomes empty. The simulation can be interrupted with the *interrupt* signal in the UNIX environment. Usually this is a control-C typed from the controlling terminal.

26. Status command

status

Output the status of all signals that are being traced, watched, or forced.

27. Step command

step

Single step the simulation. Perform one iteration of the simulation loop, processing one event from the event queue.

28. Time command

time

Output the current simulated time. Time is unitless if the current delay model is unit delay, otherwise the units depend on the resolution of the simulated time clock.

29. Toggle command

toggle signal ...

For each signal specified, if its state is ''1'' set it to ''0'' and if its state is ''0'' set it to ''1''.

30. Trace command

trace signal ...

Add the list of signals specified to those being traced. A traced signal causes an output message to be generated whenever the logical state of the signal is modified.

31. Unalias command

unalias id

Remove id from the list of aliases.

32. Untrace command

untrace signal ...

Remove the list of signals from those being traced.

33. Unwatch command

unwatch signal ...

Remove the list of signals from those being watched.

34. Watch command

watch signal ...

Add the list of signals specified to those being watched. The logical state of watched signals is output on a periodic basis under control of the output command. The position of a signal in the list of watched signals can be set by specifying a ''-number'' option before the signal name. The default position is the end of the list.

4.4. Output Format

The output format is one of the major determining factors in deciding whether any program is truly useful or is more of a hindrance than a help. With this in mind, several options have been provided to give the user some flexibility in the quantity and format of output available from the simulator.

There are two techniques that can be used to follow the logical state of signals in the circuit. The first technique, as demonstrated earlier, uses the watch and output commands to control the periodic output of signals that are of interest. Figure 11 gives a more illustrative example that demonstrates the rising and falling logical states along with the previously seen high, low, and undefined states. Note that all of the logical states other than high and low are printed in the center of the column and identified with the appropriate symbol. The column headers are repeated every 24 lines because of the -t option given in the output command, telling *lsim2* that the output is to a terminal. This insures that the column headers are on the terminal screen during the entire run. The choice of a vertical orientation for the signal traces was based on two major considerations. The first was the ability to output long printouts on continuous forms in an orderly manner. If the standard output of the *lsim2* program is redirected to a file or the printer, there is no bias toward the output being limited to a 24 line by 80 column screen. Additional signals can be watched, spreading the output across 132 columns, and the column headers will not be repeated if the -t switch is left off of the output command. The second consideration that motivated the vertical output format was a desire to maintain the portability of the program from one terminal to the next. A horizontal scroll would require cursor positioning capability on the part of the terminal that is not standard for all terminals. The vertical format does not require any non-standard terminal capabilities.

The second technique that is used to follow the logical state of signals utilizes the trace command. After a signal has been listed as being traced, every time that the state of the signal changes an output statement is printed notifying the user that the change has taken place. This technique is useful for keeping track of a large number of infrequently changing signals when watching them all would cause wraparound on an output device limited to 80 columns. A sample of the output generated by the trace command is given in Figure 12. The same inputs are used to drive the circuit as in the periodic output example earlier. The trace output is not nearly as easily readable as the periodic output, but can be useful if a large number of signals are to be monitored.

There is no restriction on the number of signals that can be watched or traced. The usefulness of the periodic output is seriously degraded, however, if enough signals are watched to cause the output lines to wrap around. There is no reason why some signals could not be watched while others are traced.

5. PROGRAMMING INTERFACE

In addition to the interactive interface, *lsim2* supports a programming interface designed to allow the generation of test vectors and running of the simulation in an automated fashion. The additional code supplied by the user is linked into *lsim2* at run time using the link interactive command and is initiated with the run command.

There are three header files that are needed when writing code to be linked to *lsim2,* they are ''types.h'', ''macros.h'', and ''lsim.ext.h''. The calls available are as follows:
1. Sntor

```
% lsim2 circuit.ls
Simulator state input from file circuit.ls
Current simulated time = 0 units.
lsim2> init variable
lsim2> set 0 a.h
lsim2> input b.h (28)0rrrr(28)1ffff p
lsim2> input clk.h (16)0(8)1(8)0 p
lsim2> watch a.h b.h di.l di.h clk.h dout.h dout.l
lsim2> output -t 1 ns
lsim2> status
a.h                     = 0 (watched)  (1)
b.h                     = 0 (watched)  (2)
di.l                    = x (watched)  (3)
di.h                    = x (watched)  (4)
clk.h                   = 0 (watched)  (5)
dout.h                  = x (watched)  (6)
dout.l                  = x (watched)  (7)
lsim2> halt 64 ns
lsim2> start
ns          1   2   3   4   5   6   7
0         |0  |0  | x | x |0  | x | x |
1         |0  |0  | x | x |0  | x | x |
2         |0  |0  | x | x |0  | x | x |
3         |0  |0  | x | x |0  | x | x |
4         |0  |0  | x | x |0  | x | x |
5         |0  |0  | x | x |0  | x | x |
6         |0  |0  | x | x |0  | x | x |
7         |0  |0  | x | x |0  | x | x |
8         |0  |0  | x | x |0  | x | x |
9         |0  |0  | x | x |0  | x | x |
10        |0  |0  | x | x |0  | x | x |
11        |0  |0  | x | x |0  | x | x |
12        |0  |0  |  1| x |0  | x | x |
13        |0  |0  |  1| x |0  | x | x |
14        |0  |0  |  1| x |0  | x | x |
15        |0  |0  |  1|0  |0  | x | x |
16        |0  |0  |  1|0  |  1| x | x |
17        |0  |0  |  1|0  |  1| x | x |
18        |0  |0  |  1|0  |  1| x | x |
19        |0  |0  |  1|0  |  1| x | x |
20        |0  |0  |  1|0  |  1| x | x |
21        |0  |0  |  1|0  |  1| x | x |
```

Figure 11. Periodic output

```
ns        1   2   3   4   5   6   7
22     |0  |0  |  1|0  |  1|0  |  1|
23     |0  |0  |  1|0  |  1|0  |  1|
24     |0  |0  |  1|0  |0  |0  |  1|
25     |0  |0  |  1|0  |0  |0  |  1|
26     |0  |0  |  1|0  |0  |0  |  1|
27     |0  |0  |  1|0  |0  |0  |  1|
28     |0  | r |  1|0  |0  |0  |  1|
29     |0  | r |  1|0  |0  |0  |  1|
30     |0  | r | f |0  |0  |0  |  1|
31     |0  | r | f |0  |0  |0  |  1|
32     |0  |  1| f |0  |0  |0  |  1|
33     |0  |  1| f |0  |0  |0  |  1|
34     |0  |  1| f |0  |0  |0  |  1|
35     |0  |  1|0  |0  |0  |0  |  1|
36     |0  |  1|0  |0  |0  |0  |  1|
37     |0  |  1|0  |0  |0  |0  |  1|
38     |0  |  1|0  | r |0  |0  |  1|
39     |0  |  1|0  | r |0  |0  |  1|
40     |0  |  1|0  | r |0  |0  |  1|
41     |0  |  1|0  | r |0  |0  |  1|
42     |0  |  1|0  | r |0  |0  |  1|
43     |0  |  1|0  | r |0  |0  |  1|
ns        1   2   3   4   5   6   7
44     |0  |  1|0  | r |0  |0  |  1|
45     |0  |  1|0  | r |0  |0  |  1|
46     |0  |  1|0  | r |0  |0  |  1|
47     |0  |  1|0  |  1|0  |0  |  1|
48     |0  |  1|0  |  1|  1|0  |  1|
49     |0  |  1|0  |  1|  1|0  |  1|
50     |0  |  1|0  |  1|  1|0  |  1|
51     |0  |  1|0  |  1|  1|0  |  1|
52     |0  |  1|0  |  1|  1| r | f |
53     |0  |  1|0  |  1|  1| r | f |
54     |0  |  1|0  |  1|  1|  1|0  |
55     |0  |  1|0  |  1|  1|  1|0  |
56     |0  |  1|0  |  1|0  |  1|0  |
57     |0  |  1|0  |  1|0  |  1|0  |
58     |0  |  1|0  |  1|0  |  1|0  |
59     |0  |  1|0  |  1|0  |  1|0  |
60     |0  | f |0  |  1|0  |  1|0  |
61     |0  | f |0  |  1|0  |  1|0  |
62     |0  | f |0  |  1|0  |  1|0  |
63     |0  | f |0  |  1|0  |  1|0  |
Simulation halted at time = 64 ns.
lsim2>
```
Figure 11. Periodic output (cont)

```
% lsim2 circuit.ls
Simulator state input from file circuit.ls
Current simulated time = 0 units.
lsim2> init variable
lsim2> set 0 a.h
lsim2> input b.h (28)0rrrr(28)1ffff p
lsim2> input clk.h (16)0(8)1(8)0 p
lsim2> trace dout.h dout.l
lsim2> status
dout.h                 = x (traced)
dout.l                 = x (traced)
lsim2> halt 64 ns
lsim2> start
Signal dout.h               modified from x to 0 at time = 22 ns.
Signal dout.l               modified from x to 1 at time = 22 ns.
Signal dout.h               modified from 0 to r at time = 52 ns.
Signal dout.l               modified from 1 to f at time = 52 ns.
Signal dout.h               modified from r to 1 at time = 54 ns.
Signal dout.l               modified from f to 0 at time = 54 ns.
Simulation halted at time = 64 ns.
lsim2>
```

Figure 12. Trace output

```
        struct signaltype *sntor(name)
                char *name;
```

Given a signal name, sntor returns a pointer to the signal record or NULL if it cannot find the signal.

2. Trace

```
        int trace(signal)
                struct signaltype *signal;
```

Given a signal pointer, trace enables the tracing of the signal. TRUE is returned if successful, FALSE is returned if the signal cannot be found or is already being traced.

3. Untrace

```
        int untrace(signal)
                struct signaltype *signal;
```

Given a signal pointer, untrace disables the tracing of the signal. TRUE is returned if successful, FALSE is returned if the signal cannot be found or is not currently being traced.

4. Set

```
        int set(signal,state,time)
                struct signaltype *signal;
                int state,time;
```

Given a signal pointer, a state, and a time, set schedules an event to set the signal to the state at the given time. The state must be from a list of provided states. The time is in picoseconds. If there is no problem with the input, the scheduling takes place and TRUE is returned, otherwise FALSE is returned.

5. Start

```
struct signaltype *start()
```

Start initiates the simulation. It returns a pointer to a signal that indicates which traced signal changed its value. NULL is returned when the simulation terminated due to a HALTRUN event or an empty event queue.

6. Halt

```
int halt(time)
        int time;
```

Given a time, halt schedules a HALTRUN event for the given time. If the input time is less than the current time, FALSE is returned. Otherwise, TRUE is returned.

7. Cont

```
struct signaltype *cont(time)
        int time;
```

Cont combines the halt and start calls into one entry point. A HALTRUN event is scheduled for crtime+time and the simulation is initiated. Cont returns a pointer to a signal that indicates which traced signal changed its value. NULL is returned when the simulation terminated due to a HALTRUN event or an empty event queue.

8. Force_s

```
int force_s(signal,state)
        struct signaltype *signal;
        int state;
```

Given a signal pointer and a state, force_s establishes a stuck-at condition on the signal. The state must be from a list of provided states.

9. Force_i

```
int force_i(comp,conn,state)
        struct comptype *comp;
        int conn,state;
```

Given a component pointer, input connection, and a state, force_i establishes a stuck-at condition on the component input. The state must be from a list of provided states.

10. Force_o

```
int force_o(comp,conn,state)
        struct comptype *comp;
        int conn,state;
```

Given a component pointer, output connection, and a state, force_o establishes a stuck-at condition on the component output. The state must be from a list of provided states.

11. Free_s

```
int free_s(signal)
        struct signaltype *signal;
```

Given a signal pointer, free_s eliminates any stuck-at conditions that existed on the signal.

12. Free_i

```
int free_i(comp,conn)
        struct comptype *comp;
        int conn;
```

Given a component pointer and input connection, free_i eliminates any stuck-at conditions that existed on the component input.

13. Free_o

```
int free_o(comp,conn)
        struct comptype *comp;
        int conn;
```

Given a component pointer and output connection, free_o eliminates any stuck-at conditions that existed on the component output.

14. Command

```
int command(str)
        char *str;
```

Command allows the programmer to use any of the interactive commands by providing a text string containing the command. TRUE is returned if the command is a valid command. FALSE is returned otherwise.

6. DEBUGGING TOOLS

As a help in debugging both the connectivity of circuits and the operation of the simulator itself, the *lsread* state debugger is available. When invoked with the following format,

```
% lsread circuit.ls
```

the circuit description specified is read in and a human readable version is output to stdout. Included in this description is the connectivity of the circuit, the state of each signal and component, the event queue, and the trace list (the list of items being traced, watched, or forced).

In addition to *lsread,* a utility called *lscnt* is available. When called as follows,

```
% lscnt circuit.ls
```

a count of all the component types present in the circuit is output to the standard output.

Appendix A

Component Types Available with Lsim2

The available components are:

and	AND gate	
pins:	Inputs:	variable (*default* 2)
	Outputs:	1
sht:	unused	
dc:	default strong high and low	
param:	unused	
Delay Suffixes:	unused	
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Bugs:	None known	

or	OR gate	
pins:	Inputs:	variable (*default* 2)
	Outputs:	1
sht:	unused	
dc:	default strong high and low	
param:	unused	
Delay Suffixes:	unused	
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Bugs:	None known	

nand	NAND gate	
pins:	Inputs:	variable (*default* 2)
	Outputs:	1
sht:	unused	
dc:	default strong high and low	
param:	unused	
Delay Suffixes:	unused	
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Bugs:	None known	

nor	NOR gate	
pins:	Inputs:	variable (*default* 2)
	Outputs:	1
sht:	unused	
dc:	default strong high and low	
param:	unused	
Delay Suffixes:	unused	
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Bugs:	None known	

xor	exclusive OR gate	
pins:	Inputs:	2
	Outputs:	1
sht:	unused	
dc:	default strong high and low	
param:	unused	
Delay Suffixes:	unused	
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Bugs:	None known	

not	NOT gate	
pins:	Inputs:	1
	Outputs:	1
sht:	unused	
dc:	default strong high and low	
param:	unused	
Delay Suffixes:	unused	
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Bugs:	None known	

buff	non-inverting buffer
pins:	Inputs: 1 Outputs: 1
sht:	unused
dc:	default strong high and low
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Bugs:	None known

dff	level sensitive D flip flop
pins:	Inputs: 2 (1 = *clock*, 2 = *data*) Outputs: 2 (1 = *true*, 2 = *complemented*)
sht:	default *st*=0*ns*, *ht*=0*ns*
dc:	default strong high and low
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Note:	Output follows input when clock is high, input is latched when clock goes low.
Bugs:	None known

etdff	positive edge triggered D flip flop
pins:	Inputs: 2 (1 = *clock*, 2 = *data*) Outputs: 2 (1 = *true*, 2 = *complemented*)
sht:	default *ht*=0*ns*, *st*=0*ns*
dc:	default strong high and low
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Note:	Input is latched when clock goes high.
Bugs:	None known

rsff	RS flip flop
pins:	Inputs: 2 (1 = *set*, 2 = *reset*)
	Outputs: 2 (1 = *true*, 2 = *complemented*)
sht:	default *st*=0*ns*, *ht*=0*ns*
dc:	default strong high and low
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Bugs:	None known

jkff	JK flip flop
pins:	Inputs: 3 (1 = *clock*, 2 = *J*, 3 = *K*)
	Outputs: 2 (1 = *true*, 2 = *complemented*)
sht:	default *ht*=0*ns*, *st*=0*ns*
dc:	default strong high and low
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Note:	Input is latched when clock goes high.
Bugs:	None known

jkrsff	JK flip flop w/ set reset
pins:	Inputs: 5 (1 = *clock*, 2 = *J*, 3 = *K*, 4 = *set*, 5 = *reset*)
	Outputs: 2 (1 = *true*, 2 = *complemented*)
sht:	default *ht*=0*ns*, *st*=0*ns*
dc:	default strong high and low
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Note:	JK inputs are latched when clock goes high, RS inputs are asynchronous.
Bugs:	None known

passtran	n channel bidirectional pass transistor
pins:	Inputs: 2 (1 = *gate* , 2 = *source*) Outputs: 1 (*drain*)
sht:	unused
dc:	function of input signal
param:	unused
Delay Suffixes:	unused
Default Delay:	(0,0,0 $ 0,0,0 $ 1,1,1 $ 1,1,1)ns
Note:	Input #2 and output #1 are treated the same, both are actually i/o connections. This is not an inertial gate, spike errors are not checked.
Bugs:	None known

pptran	p channel bidirectional pass transistor
pins:	Inputs: 2 (1 = *gate* , 2 = *source*) Outputs: 1 (*drain*)
sht:	unused
dc:	function of input signal
param:	unused
Delay Suffixes:	unused
Default Delay:	(0,0,0 $ 0,0,0 $ 1,1,1 $ 1,1,1)ns
Note:	Input #2 and output #1 are treated the same, both are actually i/o connections. This is not an inertial gate, spike errors are not checked.
Bugs:	None known

unptran

n channel unidirectional pass transistor

pins:	Inputs: 2 (1 = *gate* , 2 = *source*)
	Outputs: 1 (*drain*)
sht:	unused
dc:	function of input signal
param:	unused
Delay Suffixes:	unused
Default Delay:	(0,0,0 $ 0,0,0 $ 1,1,1 $ 1,1,1)ns
Note:	This component will not help detect sneak paths, it is unidirectional only.
Bugs:	None known

upptran

p channel unidirectional pass transistor

pins:	Inputs: 2 (1 = *gate* , 2 = *source*)
	Outputs: 1 (*drain*)
sht:	unused
dc:	function of input signal
param:	unused
Delay Suffixes:	unused
Default Delay:	(0,0,0 $ 0,0,0 $ 1,1,1 $ 1,1,1)ns
Note:	This component will not help detect sneak paths, it is unidirectional only.
Bugs:	None known

resistor	resistor
pins:	Inputs: 1
	Outputs: 1
sht:	unused
dc:	default weak high and low
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Note:	Input and output are treated the same,
	both are actually i/o connections.
	This is not an inertial gate, spike errors are not checked
Bugs:	None known

linedelay	line delay component
pins:	Inputs: 1
	Outputs: 1
sht:	unused
dc:	function of input signal
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Note:	This is not an inertial gate, spike errors are not checked
Bugs:	None known

arbiter	arbiter component
pins:	Inputs: 2
	Outputs: 2
sht:	unused
dc:	default strong high and low
param:	unused
Delay Suffixes:	unused
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns
Bugs:	Variable delay not tested. Spike errors might be erroneously reported.

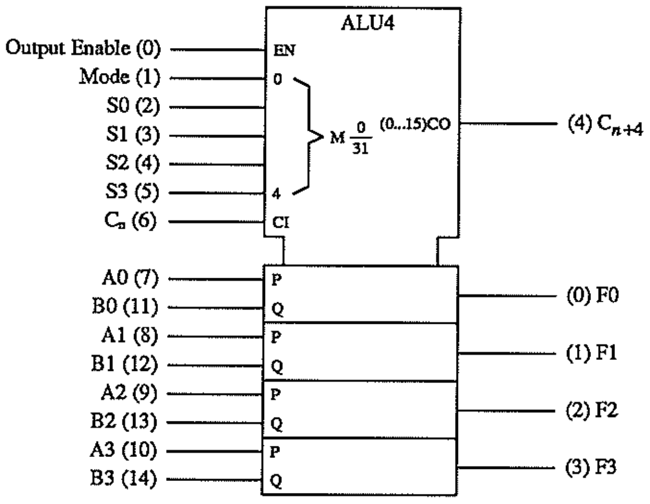
alu — Arithmetic / Logic Unit

pins:	Inputs:	variable (2**width* + 7)
	Outputs:	variable (*width*)
sht:	unused	
dc:	default strong high and low	
param:	**md**	output enable mode $(0 \rightarrow 7)$
	wid	data path width $(\leq \log_2 MAXINT)$
	sz	unused
	msc	unused
	clkopt	unused
	clropt	unused
Delay Suffixes:	eo	output enable delay
	do	data in to data out delay
	fo	function in to data out delay
	co	carry in to data out delay
	mo	mode in to data out delay
	dc	data in to carry out delay
	fc	function in to carry out delay
	cc	carry in to carry out delay
	mc	mode in to carry out delay
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation.	
	MAXINT is the largest representable integer on the computer in use.	
Bugs:	None known	

Function Code	Operation	
	Logic mode	Arithmetic mode
0	$F = \overline{A}$	$F = A \textit{ plus } C_n$
1	$F = \overline{A + B}$	$F = A + B \textit{ plus } C_n$
2	$F = \overline{A}B$	$F = A + \overline{B} \textit{ plus } C_n$
3	$F = 0$	$F = \textit{minus } 1 \, (\textit{2's comp}) \textit{ plus } C_n$
4	$F = \overline{AB}$	$F = A \textit{ plus } A\overline{B} \textit{ plus } C_n$
5	$F = \overline{B}$	$F = (A + B) \textit{ plus } A\overline{B} \textit{ plus } C_n$
6	$F = A \textit{ xor } B$	$F = A \textit{ minus } B \textit{ minus } 1 \textit{ plus } C_n$
7	$F = A\overline{B}$	$F = A\overline{B} \textit{ minus } 1 \textit{ plus } C_n$
8	$F = \overline{A} + B$	$F = A \textit{ plus } AB \textit{ plus } C_n$
9	$F = \overline{A \textit{ xor } B}$	$F = A \textit{ plus } B \textit{ plus } C_n$
10	$F = B$	$F = (A + \overline{B}) \textit{ plus } AB \textit{ plus } C_n$
11	$F = AB$	$F = AB \textit{ minus } 1 \textit{ plus } C_n$
12	$F = 1$	$F = A \textit{ plus } A^* \textit{ plus } C_n$
13	$F = A + \overline{B}$	$F = (A + B) \textit{ plus } A \textit{ plus } C_n$
14	$F = A + B$	$F = (A + \overline{B}) \textit{ plus } A \textit{ plus } C_n$
15	$F = A$	$F = A \textit{ minus } 1 \textit{ plus } C_n$

* (left shifted).

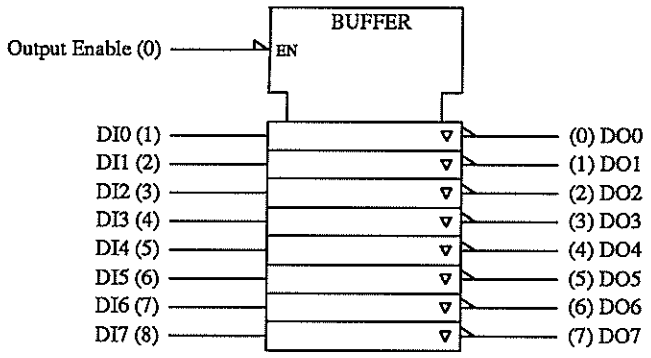
ALU Functions and Codes.



```
my_alu = (alu,pins=(15,5),param=(md=0,wid=4,sz=0,msc=0));
```

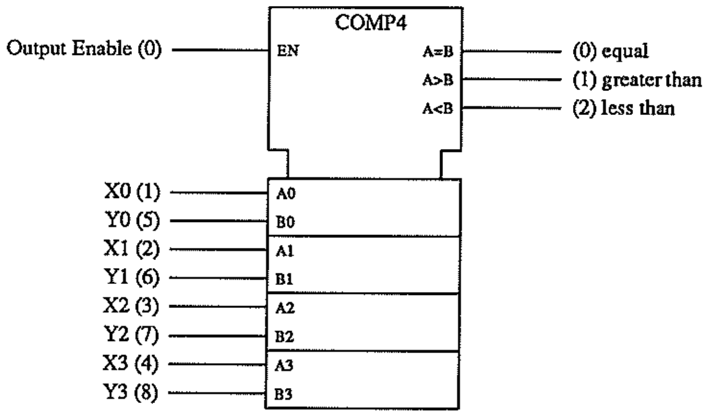
buffer

buffer	generic buffer/driver	
pins:	Inputs:	variable (*width* + 1)
	Outputs:	variable (*width*)
sht:	unused	
dc:	default strong high and low	
param:	**md**	output enable mode $(0 \rightarrow 7)$
	wid	data path width ($\leq$ *MAXINT*)
	sz	unused
	msc	unused
	clkopt	unused
	clropt	unused
Delay Suffixes:	unused	
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation.	
	MAXINT is the largest representable integer on the computer in use.	
Bugs:	None known	



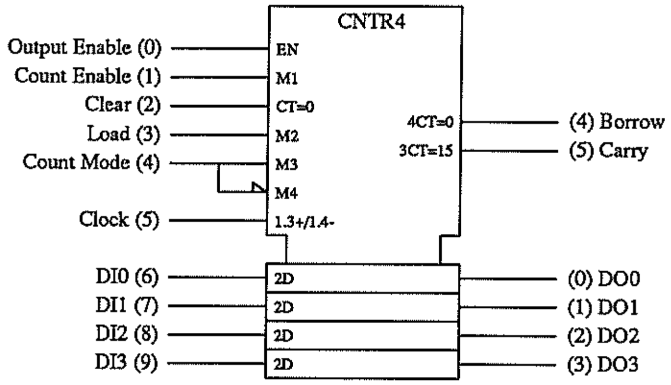
my_buffer = (buffer,pins=(9,8),param=(md=7,wid=8));

cmp	magnitude comparator	
pins:	Inputs:	variable $(2*width+1)$
	Outputs:	3
sht:	unused	
dc:	default strong high and low	
param:	**md**	output enable mode $(0 \rightarrow 7)$
	wid	data path width $(\leq \log_2 MAXINT)$
	sz	unused
	msc	unused
	clkopt	unused
	clropt	unused
Delay Suffixes:	eo	output enable delay
	do	data in to data out delay
Default Delay:	(1,1,1 \$ 1,1,1 \$ 0,0,0 \$ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation.	
	MAXINT is the largest representable integer on the computer in use.	
Bugs:	None known	



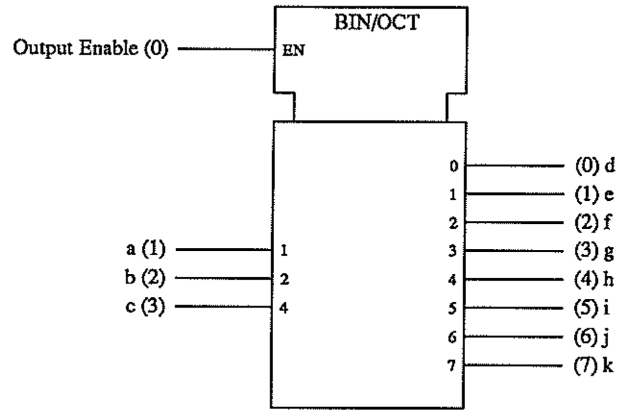
```
my_cmp = (cmp,pins=(9,3),param=(md=0,wid=4,sz=0,msc=0));
```

cntr	generic up/down counter	
pins:	Inputs:	variable $(width + 6 \text{ or } width + 7)$
	Outputs:	variable $(width + 2)$
sht:	default $ht = 0ns$, $st = 0ns$	
dc:	default strong high and low	
param:	**md**	output enable mode $(0 \rightarrow 7)$
	wid	data path width $(\leq \log_2 MAXINT)$
	sz	count modulus $(\leq MAXINT)$
	msc	type ‖ load_polarity (0=*active high*, 1=*active low*) ‖ load_type (0=*asychronous*, 1=*synchronous*)
	clkopt	clock_polarity (0=*active high*, 1=*active low*) ‖ 0 ‖ clocks (1 *or* 2)
	clropt	count_enable_polarity (0=*active high*, 1=*active low*) ‖ clear_polarity (0=*active high*, 1=*active low*) ‖ clear_type (0=*asynchronous*, 1=*synchronous*)
Delay Suffixes:	eo	output enable delay
	mo	mode in to data out delay
	mc	mode in to carry out delay
	ko	clock in to data out delay
	kc	clock in to carry out delay
	kb	clock in to borrow out delay
	co	clear in to data out delay
	cc	clear in to carry out delay
	cb	clear in to borrow out delay
	lo	load in to data out delay
	lc	load in to carry out delay
	lb	load in to borrow out delay
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation. ‖ is used to denote concatenation. *MAXINT* is the largest representable integer on the computer in use.	
Bugs:	None known	

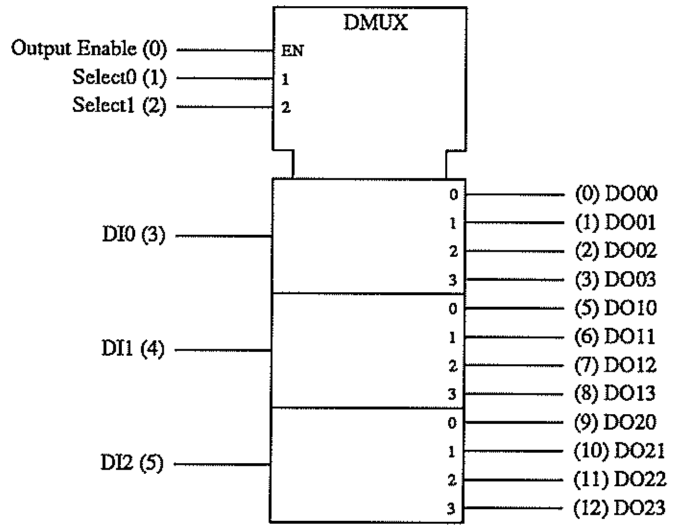


my_cntr = (cntr,pins=(10,6),param=(md=0,wid=4,sz=16,msc=000,clkopt=000,clropt=000));

dec	decoder/demultiplexer	
pins:	Inputs:	variable ($\log_2 functional\_size + width + 1$)
	Outputs:	variable ($width * \log_2 functional\_size$)
sht:	unused	
dc:	default strong high and low	
param:	**md**	output enable mode ($0 \rightarrow 7$)
	wid	data path width ($\leq \log_2 MAXINT$)
	sz	functional size (number of select lines) ($\leq MAXINT$)
	msc	type (0=*decoder*, 1=*demultiplexer*)
	clkopt	unused
	clropt	unused
Delay Suffixes:	eo	output enable delay
	do	data in to data out delay
	so	select in to data out delay
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation.	
	MAXINT is the largest representable integer on the computer in use.	
Bugs:	None known	

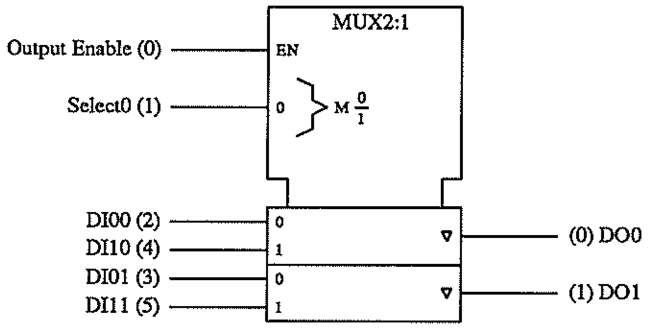


```
my_dec = (dec,pins=(4,8),param=(md=0,wid=1,sz=3,msc=0));
```

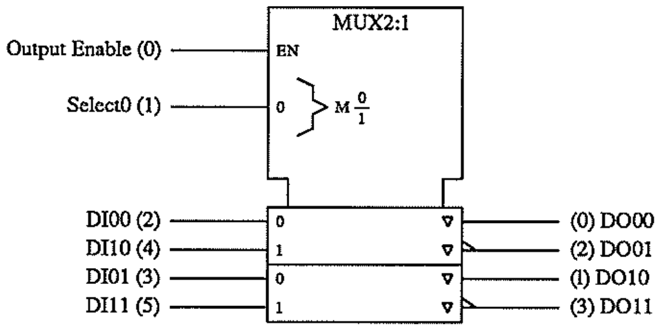


```
my_demux = (dec,pins=(6,12),param=(md=0,wid=3,sz=2,msc=1));
```

mux	multiplexer	
pins:	Inputs:	variable ($width*functional\_size + \log_2 functional\_size + 1$)
	Outputs:	variable ($width$)
sht:	unused	
dc:	default strong high and low	
param:	**md**	output enable mode ($0 \rightarrow 7$)
	wid	data path width ($\leq \log_2 MAXINT$)
	sz	functional size (number of select lines) ($\leq MAXINT$)
	msc	output rails (1 *or* 2)
	clkopt	unused
	clropt	unused
Delay Suffixes:	eo	output enable delay
	do	data in to data out delay
	so	select in to data out delay
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation.	
	MAXINT is the largest representable integer on the computer in use.	
Bugs:	None known	

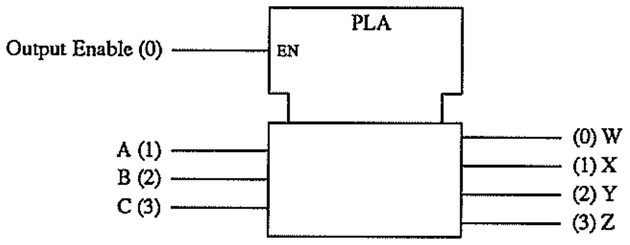


my_mux = (mux,pins=(6,2),param=(md=1,wid=2,sz=1,msc=1));



my_mux = (mux,pins=(6,4),param=(md=1,wid=2,sz=1,msc=2));

pla	programmable logic array	
pins:	Inputs:	variable (*input_width* + 1)
	Outputs:	variable (*output_width*)
sht:	unused	
dc:	default strong high and low	
param:	**md**	output enable mode $(0 \rightarrow 7)$
	wid	input_width ($\leq$ *MAXINT*)
	sz	output_width ($\leq$ *MAXINT*)
	msc	terms ($\leq$ *MAXINT*)
	clkopt	unused
	clropt	unused
Delay Suffixes:	eo	output enable delay
	do	data in to data out delay
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation. Product of input_width and terms and the product of output_width and terms must also each be less than *MAXINT* *MAXINT* is the largest representable integer on the computer in use.	
Bugs:	None known	



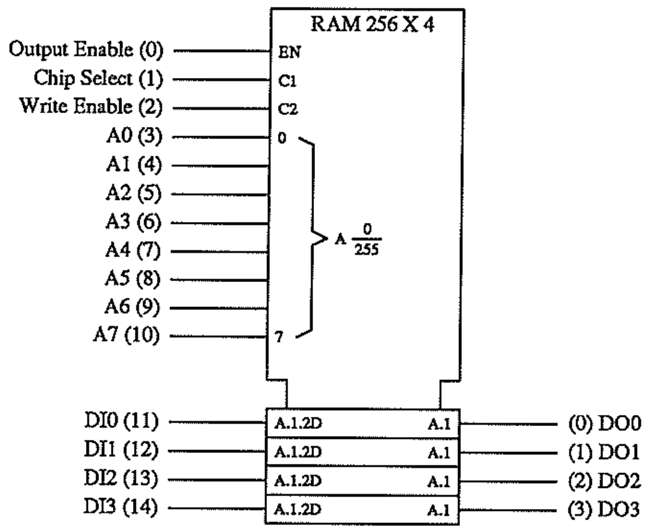
my_pla = (pla,pins=(4,5),param=(md=0,wid=3,sz=4,msc=5));

mask = (PLA,W = !C + A*B, X = A*!C, Y = B*C, Z = !A*!B);

A	B	C	W	X	Y	Z
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	1	0	1	0	0	0
0	1	1	0	0	1	0
1	0	0	1	1	0	1
1	0	1	0	0	0	1
1	1	0	1	1	0	0
1	1	1	1	0	1	0

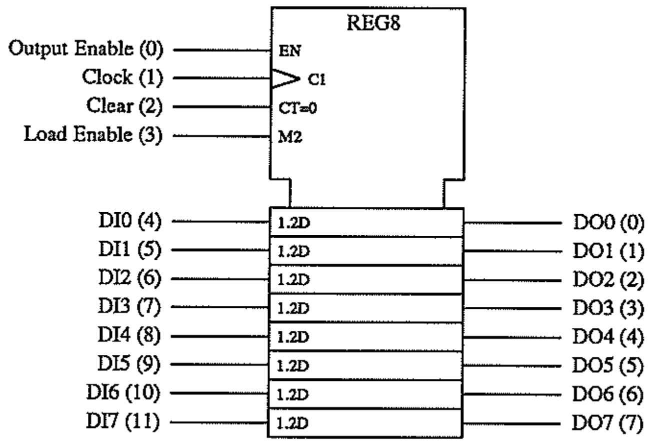
Truth Table for *my_pla*.

ram	static RAM/ROM	
pins:	Inputs:	variable ($\log_2 size + 3 + width$ *or* $\log_2 size + 1$)
	Outputs:	variable ($width$)
sht:	default $ht = 0ns$, $st = 0ns$	
dc:	default strong high and low	
param:	**md**	output enable mode ($0 \rightarrow 7$)
	wid	data path width ($\leq MAXINT$)
	sz	memory size (number of address lines) ($\leq \log_2 MAXINT$)
	msc	type (0=*ROM*, 1=*RAM*) ‖ chip_select_polarity (0=*active high*, 1=*active low*) ‖ write_enable_polarity (0=*active high*, 1=*active low*)
	clkopt	unused
	clropt	unused
Delay Suffixes:	eo	output enable delay
	do	data in to data out delay
	ao	address in to data out delay
	co	chip select in to data out delay
Default Delay:	(1,1,1 $ 1,1,1 $ 0,0,0 $ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation. ‖ is used to denote concatenation. *MAXINT* is the largest representable integer on the computer in use.	
Bugs:	None known	

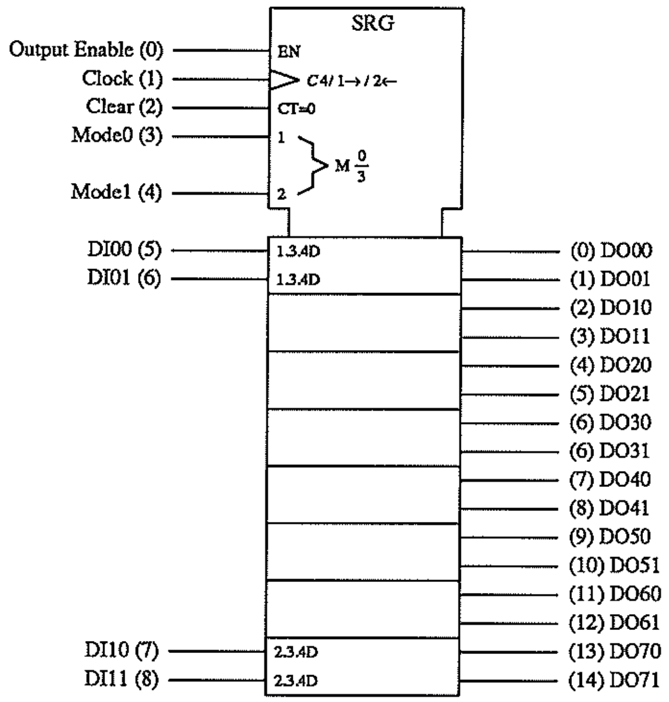


```
my_ram = (ram,pins=(15,4),param=(md=0,wid=4,sz=8,msc=100));
```

reg	generic register component	
pins:	Inputs:	variable (*width* + 5 *or width* + 6)
	Outputs:	variable (*see note*)
sht:	default $ht = 0ns$, $st = 0ns$	
dc:	default strong high and low	
param:	**md**	output enable mode $(0 \rightarrow 7)$
	wid	data path width $(\leq \log_2 MAXINT)$
	sz	shift depth $(\leq MAXINT)$
	msc	type (1=*shift register*, 0=*register/latch*) ‖ outconfig (*see note*) ‖ inconfig (*see note*)
	clkopt	clock_polarity (0=*active high*, 1=*active low*) ‖ clock_type (0=*level sensitive*, 1=*edge triggered*) ‖ clocks (1 *or* 2)
	clropt	clear_polarity (0=*active high*, 1=*active low*) ‖ clear_type (0=*asynchronous*, 1=*synchronous*)
Delay Suffixes:	eo	output enable delay
	do	data in to data out delay
	mo	mode control in to data out delay
	co	clock active to data out delay
Default Delay:	(1,1,1 \$ 1,1,1 \$ 0,0,0 \$ 0,0,0)ns	
Note:	See appendix B for output enable mode explanation. ‖ is used to denote concatenation. inconfig and outconfig take values between 0 and 3. $0 \rightarrow$ parallel inputs (outputs) $1 \rightarrow$ right serial inputs (outputs) only $2 \rightarrow$ left serial inputs (outputs) only $3 \rightarrow$ right and left serial inputs (outputs) only *MAXINT* is the largest representable integer on the computer in use.	
Bugs:	None known	



my_reg = (reg,pins=(12,8),param=(md=0,wid=8,sz=0,msc=000,clkopt=011,clropt=00));

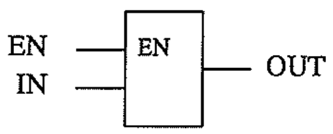


```
my_sh_reg = (reg,pins=(12,8),param=(md=0,wid=2,sz=8,msc=103,clkopt=011,clropt=00));
```

Appendix B

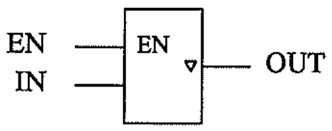
Output Enable Modes

All of the MSI type components in the *lsim2* library have as an option a selectable output enable mode. There are a total of eight modes avaliable identified by the values 0 through 7. For each of the modes, a simple buffer will be shown to illustrate the meaning of the mode. In addition, a truth table will be presented to further clarify the operation of the output enable input.



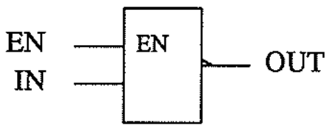
en	in	out
0	0	0
0	1	0
1	0	0
1	1	1

Mode 0. Active High Control / Active High Output.



en	in	out
0	0	Z
0	1	Z
1	0	0
1	1	1

Mode 1. Active High Control / Active High Tri-state Output.



en	in	out
0	0	1
0	1	1
1	0	1
1	1	0

Mode 2. Active High Control / Active Low Output.