Logic Simulator

Abstract

A gate-level logic simulation and design tool is described. Interconnections between gates are described using a compact notation called Logic Description Language (LDL). Gate and wire delays may be specified with 0.001 nSec (picoSecond) resolution. A graphical map of every gate and its current state is provided. The mouse can be used to examine input and output details of every gate. Selected gates can be displayed on oscilloscope-like traces. These traces may be zoomed, panned and measured for values such as intervals, pulse widths and periods.

A stimulus generator may be used to provide synthetic inputs from external logic blocks. Complex conditionals may be used to generate signal values and detect error conditions and timing violations.

A complete wire and gate list for the design may be generated, and used to create VHDL, Verilog or RTL inputs for use by other design tools.

The Simulator addresses the three fundamental shortcomings of existing tools:

1. **Quick and Accurate Description** of design elements,
2. **Visual Answers** to fundamental questions about the design, and
3. **Direct Comparison** of alternative designs.

The entire simulator is a single HTML / JavaScript file which may be run on any modern web browser.

Contents

Logic Simulator Overview
Detailed Feature Descriptions - (TBD)
User Guide ($UserGuide) - (TBD)

- Getting Started

Logic Description Language

- Comments
- Conditional Compilation
- Statements

Symbol Table Concepts

- Symbol Substitution
- Indentation and Iterative Generation
- Parallel Assignments

Gate Descriptions

- Gate Delays *vs.* Rise and Fall Times
- Power Consumption

Macro Logic

- Macro Syntax

Overrides and Redefinition of Gates
Gate Maps - (TBD)
Schematic Synthesis - (TBD)
Signal Traces - (TBD)
Stimulus State Machines - (TBD)
Fault Tolerance

- Monte Carlo Delay Simulation
- Boundary Scan - (TBD)
- Noise Injection - (TBD)

Collaboration

- Cloud Storage

- Local Files
- Project Documentation

Design Guidelines

- Design for Testability
- Symbol Names
- Title and Note
- Naming Conventions

Simulator Configuration

Why Another Design Automation Tool?

The world of Electronic Design Automation is rife with comprehensive, mature, industry standard products targeted for all phases of electronic, logic and chip design. Large development companies, maintenance contracts, skilled professionals and training programs abound. So why did I develop this tool?

- “What I tell you three times is true.” - Lewis Carroll, *The Hunting of the Snark*

I intend to provide an alternative, rapid, technology to describe, demonstrate and test logic designs. The description language is completely orthogonal to traditional RTL, VHDL, Verilog or WYSIWIG tools. Emphasis is on documentation, visualization and accuracy. Calculations can simulate logic blocks at all levels of complexity and time scale to provide designers and managers with sanity checks and increased confidence in a design. These “back of the napkin” results can be used as a starting point for more traditional design tools, and to provide independent validation tests.

I expect that a design can be described as:

- Descriptive, literate design narrative
- Stimulus and Response via the Stimulus State Machine feature
- Gate-level logic, and
- RTL, VHDL, Verilog, etc. using other EDA tools.

The use of multiple, completely independent methodologies to describe a target project will improve the quality of the design and help prevent “shared misunderstanding” errors. When I “tell you three times”, I try to use different words in different languages. Again, per Lewis

Carroll: "*I said it in Hebrew - I said it in Dutch - I said it in German and Greek*". This helps ensure that we can discover the difference between a snark and a boojum as early as possible.

"Won't it take too long or cost too much to do parallel designs?" Let us consider the current crop of EDA tools in more detail.

Feature	...
Rife with Tools	Incompatible
Comprehensive	Bloated
Mature	Legacy Architecture
Industry Standard	Designed by Committee
All Phases of Design	Feature Accretion
Large Development Companies	Expensive
Maintenance Contracts	Buggy
Skilled Professionals	Vested Interests
Training Programs	Steep Learning Curve

Now, after that (admittedly, rather snide) recharacterization of my initial paragraph (and the industry as a whole), let us examine my current offering to see if I am proposing time and money well spent.

Logic Simulator Overview

We provide a number of features to assist in the block-level and gate-level design of complex logic.

- Compact notation to replace VHDL, Verilog and RTL descriptions in most cases
    - Explicit names for all signals
    - Complete wire-lists showing all gate connections
    - Full reduction to NAND logic
    - Precision timing and delays at up to picoSecond accuracy

    - Direct indication of Fan-Out for every signal

- Gate-level, picoSecond accurate Logic Simulation
    - Variable-speed Run mode, including single-step
    - Dynamic real-time displays
    - Gate delays based on wire length and fan-out loading

- Display map of every gate and its current state
    - Mouse selection showing input states and fan-out
    - Dynamic colors indicate recent state changes

- Oscilloscope-like display traces showing history of selected gates
    - Traces allow wild-card grouping to show bus status
    - Mouse selection to analyze exact state at a given time
    - Precision time interval measurement, and pulse width and period display
    - Dynamic zoom and pan through trace history

- Unique syntax describing Stimulus State Machines which are used for
    - No need for external C or Python simulation scripts
    - Stimulus generation
    - Simulation of external logic blocks
    - User interface buttons and lights
    - Assertions and Error Detection
    - Signal Trace Annotation
    - Event Logging

- Support for direct comparison of alternative designs

- Logging of event conditions

- Cloud-based shared-source structure for collaborative design

Detailed Feature Descriptions

```
(TBD)
```

User Guide

Getting Started

```
(TBD)
```

Logic Description Language

The Logic Description Language(LDL) is an ASCII text-based, line-oriented language.

Comments

LDL is considered **literate**, in that descriptive comments are the predominant, default mode. All lines that have text which begins in column one are considered comments and are ignored by the processor. Additional end-of-line comments begin with **//** or **!!**, and traditional C-like multi-line comments bracketed by **/*** and ***/** are also supported.

The comment processing step is completely unaware of literals or quoted strings. If you insist on trying to put things that look at comment markers in literals, write them like this: **/\/**, **!\!**, /* and *\/ using the **** before the second character.

Conditional Compilation

Conditional Compilation is used to allow sections of source code to be selectively included or removed from the simulation. Special syntax for multi-line comments is used. **/*$ name** … ***/** indicates that an option **name** will be tested. This creates an option check box called **name** on the user interface which, if checked, will cause the statements up to the end of the comment to be processed instead of being ignored. There is no **else** option. Therefore, we also support the inverse form: **/*$! name** … ***/** .

conditional form	description
/* … */	Multi-line Comment. Do not process … statements
/*$ name … */	Process statements … if option check box **name** is checked
/*$! name … */	Process statements … if option check box **name** is not checked
/*$ name:true */	Initialize option check box **name** to checked
/*$ name:false */	Initialize option check box **name** to not-checked

Multi-line comments may **not** be nested.

This feature approximates the conditional compilation provided to C-like languages by the **#define**, **#ifdef** and **#endif** preprocessor directives.

Statements
Language statements begin after column one and are indentation-dependent. That is, the number of leading spaces is used to establish statement groups. This is in lieu of the bracketing symbols such as **{** and **}**, or **begin** and **end** commonly used in other languages.

Each line of source may contain at most one statement. There is no statement delimiter. Parameters within a statement are separated by the **;** (semi-colon). Sub-parameters are separated by the **,** (comma).

There are several types of statements:

statement type	example
Option Block Begin	`/* $name`
Option Block End	`*/`
Symbol Assignment	`: symbol=value`
Sequential Assignment	`: symbol=val1,val2,val3`
Parallel Assignment	`: symbol1=valA,valB,valC; symbol2=valX,valY,valZ`
Gate Description	`name; gate=NOT; a=inputSignal`
Gate Macro Start	`[[ name; param1; param2`
Gate Macro End	`]]`
Gate Map Format	`@ numberOfColumns, columnWidth, gateSize`

Gate Map Column	`@ col`
Gate Map Row	`@`
Stimulus Machine Begin	`{{`
Stimulus Machine End	`}}`
Stimulus Machine State	`** name`
Stimulus Machine Condition	`?? condition`
Stimulus Machine Operation	`>> operation:parameter`
Stimulus Machine Assignment	`>> name:=value`

The use of explicit markers for statement type at the beginning of most lines improves clarity and eliminates most misunderstandings during casual source review. In general, block-style statement groups may not be nested, further reducing errors.

Symbol Table Concepts

Programming languages commonly use the concept of “variables” to allow convenient, meaningful names to be used as placeholders for actual “values” to be used by the program. The correspondence between variable “names” and “values” are contained within a structure known as the Symbol Table. Using a Symbol Table allows any particular variable “name” to be assigned different “values” as needed at different points in the program.

The symbol replacement mechanism performs a similar function to the macro pre-processor used in C-like languages.

In the Logic Simulator, the Symbol Table is used only during the process of converting an input source program into the gates, Gate Maps, wire connection lists and Stimulus State Machine structures used for the actual simulation. This “build a symbol” technique is used as a compact notation to describe the creation of the gate names and connections used in the actual simulation.

LDL uses a hierarchial symbol table while processing the source input program.
Processing of the source generates

1. A Graphical Gate Map,
2. A Gate Connection List, and
3. A Stimulus State Machine list.

Lexicographic levels are explicitly indicated by indentation of the source.
The specific lex level is specified by the number of steps of indentation, not the actual distance (number of leading spaces). Thus, any conveninent indentation style can be used for readability - so long as each lex level lines up consistently. The following (contrived) examples generate identical results.

```
: foo=bar, baz
  : count=47, 49
    signal foo count; gate=NAND; a=foo; b=signal count
signal foo count; gate=NAND; a=foo; b=signal count

: foo=bar, baz
      : count=47, 49
            signal foo count; gate=NAND; a=foo; b=signal count
signal foo count; gate=NAND; a=foo; b=signal count
```

Caveat: One must be aware of all of the defined symbols in the current lexical group in order to prevent “accidental” use of a defined symbol when a literal is intended. In collaborative designs (or simply to improve clarity) a Hungarian Notation convention may be adopted. For example, all defined symbols could begin with **&** (ampersand), and all iterative symbols could begin with **&&** (double ampersand). This would be a design convention and not enforced by the language. In general, such contrivances should not be necessary.

Symbol Substitution

Consider the two examples above. Each has four source lines, both generate five gates and are equivalent to the following:

```
signalbar47; gate=NAND; a=bar; b=signal47
signalbar48; gate=NAND; a=bar; b=signal48
signalbaz47; gate=NAND; a=baz; b=signal47
signalbaz48; gate=NAND; a=baz; b=signal48
signalbazcount; gate=NAND; a=baz; b=signalcount
```

Indentation and Iterative Generation

The first statement is a symbol table assignment (begins with a **:** colon) at the outermost lex level. The symbol **foo** will take on two consecutive values **bar** and **baz**. This is how we create loops: a sequential-value assignment followed by an indented block of statements that are repeated once for each value.

The second statement is a similar multi-value assignment, creating the symbol **count** at the next lex level. This nested loop structure steps through two values of **count** (**47** and **49**) for each of the values of **foo**.

The third statement is a description of a new gate. The name of the gate (and the name of its output signal) is given by the text string before the first **;** (semi-colon). The “words” **foo** and **count** are substituted to compose the actual Gate Description statement.

The fourth statement is indented to the same level as the definition of **foo** and therefore is outside of either of the loops. It also describes a new gate. The symbol **foo** retains its most recent value (**baz**), and the “word” **count** is not defined in this outer lex level and is therefore treated as a literal.

One of the most important features of any logic design is the ability to create groups of similar logic elements such as busses or registers that all have unique but descriptive names. For example, **reg0**, **reg1**, … **regF**. The need to concatenate elements to create signal names is so common that it is a primary feature of LDL.

When processing a statement, every “word” (printable characters separated by spaces) may be either a literal part of a name or a reference to a previously-defined symbol. For each “word” the symbol table is examined, beginning with the current lex level and working toward the outer level. If a defined symbol is found whose name matches the word, the value is substituted. If the “word” is not found in the symbol table, it is treated as a literal.

After all symbol substitution is completed, spaces are removed from the statement. This effectively concatenates the static (literal) and the dynamic parts of the signal name.

Symbolic name generation is so important and so common that it is critical that the syntax used should be clear, compact and not error-prone. Traditional languages usually feature 1960’s style techniques such as quoted-strings for literals and explicit concatenation

operators such as **+** (plus). This is labor-intensive, error-prone, and tends to obscure the designer's intent. Making "word" lookup and concatenation the default behavior enhances to clarity and accuracy of the design.

Parallel Assignment

Another common design requirement is to be able to create gates connected to adjacent elements (of a register or bus, for example). This quick description generates the first level of a parity generator for an eight-bit register.

```
: inA=0,2,4,6; inB=1,3,5,7
    parity inA inB; gate=XOR; a=reg inA; b=reg inB
```

This uses the same symbol table value assignment, substitution and looping described above. The difference is that successive values are assigned to both inA and inB for each iteration. The resulting four gate descriptions woud be:

```
parity01; gate=XOR; a=reg0; b=reg1
parity23; gate=XOR; a=reg2; b=reg3
parity45; gate=XOR; a=reg4; b=reg5
parity67; gate=XOR; a=reg6; b=reg7
```

The number of elements in the list assigned to each of the symbols should be the same. Parallel assingments are not restricted on number of symbols or number of iterative elements per symbol.

To complete the example, a full eight-bit parity generator could be described as follows:

```
: inA=0,2,4,6; inB=1,3,5,7
    parity inA inB; gate=XOR; a=reg inA; b=reg inB
: inA=01,45; inB=23,67
    parity inA inB; gate=XOR; a=reg inA; b=reg inB
parity01234567; gate=XOR; a=parity0123; b=parity4567
```

The complete generator tree uses seven XOR gates in three levels (of 4, 2 and 1) and computes the result in $\boldsymbol{O}(\log_2 n)$ where n is the number of bits in the register.

An alternative (naive) design also uses seven XOR gates:

```
parity 1; gate=XOR; a=reg 0; b=reg 1
: inA=1,2,3,4,5,6; inB=2,3,4,5,6,7
    parity inB; gate=XOR; a=parity inA; b=reg inB
```

This is a (very slow) serial chain of XOR gates which computes the result in ***O***(n) where n is the number of bits in the register.

Gate Descriptions

The Logic Description Language is used to create a gate-by-gate description of the logic and interconnections of the design. Every gate is named by its single output signal. The output signal can be thought of as the “value” of the gate at any time.

Output signals from a gate are used as inputs to other gates. This creates a Fan-Out situation which allows one driver and multiple receivers.

Every signal is sourced by exactly one gate and drives one or more inputs. This explicitly disallows tri-state outputs or wired-OR connections as these are considered poor practice for modern designs.

We are most concerned with NAND gates for conventional logic designs, although we natively allow any of the following:

Inputs:	**2-9**	**2-9**	**2**	**1**
	AND	OR	XOR	BUF
	NAND	NOR	XNOR	NOT

Inverters (**NOT**) and buffers (**BUF**) have one input named **a**.
XOR and **XNOR** have exactly two inputs named **a** and **b**.
AND, **NAND** and **OR** types have two to nine inputs, named **a b c d e f g h i**. All input names must be consecutive. Input names must be lower-case letters.

Each gate description is a line of source code such as this:

```
gateName; gate=NAND; a=inputA; b=inputB

gateName#; gate=NOT; a=gateName
```

Gate Delays *vs.* Rise and Fall Times

This Simulator is intended for the modeling of high-speed, chip level designs. Traditional concerns for a signal's rise and fall times are based on the driver pumping charge into a *bulk capacitance* and the duration of the period of ambiguity at the connected input devices. We eschew designs involving tri-state logic, open drain (open-collector) wired-OR signals and bipolar devices. Thus, any differences between rise- and fall-times become irrelevant.

For high-speed designs using CMOS devices we are more concerned with the signal delays due to transmission-line effects and *distributed impedance*. Instead of attempting analog modeling, this simulator introduces scaled delays between the outputs and inputs of otherwise ideal logic gates. Thus, the waveforms displays for a given gate are viewed from the standpoint of an ideal probe at the output of an ideal gate.

A signal transition seen at the output of a driving gate will not affect the state of a driven gate for some interval into the future. We allow several elements to influence these delays in order to adapt to real-world situations.

Delay Element	Description
Inherent Gate Delay	Switching time driving standard FO4 load
Additional Load	Greater Fan Out slows transitions
Manufacturing Variation	Randomized Delay Variation
Wire Length	Unique additional delay per input

Each of these elements may be adjusted as part of the design specification.

Power Consumption

In general, the power consumption of modern logic circuits can be divided into *static* and *dynamic* components. For CMOS circuitry this can be roughly analyzed given the parameters in the following table. Note the distinction between CMOS FET Gates and Logic Gates.

Parameter	Static	Dynamic
FET Gate Area	Y	Y
Number of FET Gates	Y	Y
Number of FET Gate State Changes		Y
Operating Voltage	Y	
Signal Voltage Range		Y

In order to assist with these calculations for logic gate-level designs, the simulator provides several useful statistics.

Parameter	Description
Number of Logic Gates	Total Logic Gates (output signals) synthesized in the design
Number of FET Gates	Total Logic Gate Inputs (times 2) synthesized in the design
Number of Transitions	FET Gates that change state during a selected time interval
Running Average Transitions	Average Dynamic Power Consumption
Running Peak Transitions	Peak Dynamic Power Consumption

One primary benefit of these simulation results is the ability to directly compare and optimize competing designs. The design that meets the goal with fewest gates and fewest, lowest-speed clock transitions should be preferred.

The Running Average and Running Peak values assist in addressing noise considerations. Strong Peak values are indicative of a design that is likely to couple noise onto the power rails, as well as contributing to radiated emissions.

It is up to the designer to know the details of process and geometry in order to properly scale these guides into real-world estimates.

Macro Logic

Macros provide a mechanism to name and define common logic blocks. Several library macros are provided for commonly used elements.

Internal support for **AND OR XOR NOR** and **XNOR** are provided for quick-and-dirty simulations. Higher fidelity simulation and accurate wiring lists would normally expect these functions to be implemented in pure **NAND** logic. Library macros are provided for this purpose.

Note that the library macros require an explicit number of inputs. The macro parameter substitution uses **A=** and **B=** in Upper Case. Native gates use lower case input names and automagically handle up to nine inputs.

The designer is expected to make a decision as to whether the full fidelity implementation is needed for the particular situation. Additional macros can be created to build blocks for larger numbers of inputs for simple gate functions.

Macros: Full fidelity reduction to NAND-only logic

Native Gates: Easy preliminary design and faster simulation

The library macros that provide reduction to NAND logic will create multiple-input NAND gates. This is the expected behavior for CMOS logic design targeted for FPGA or ASIC hardware.

By convention, gates that are created within a macro are given names ending with **–a**, **–b**, etc. In some cases, more descriptive suffixes are used. The primary output(s) of the logic block will be given the name without any suffix. Flip-flops with an inverted output will provide a gate with the **#** suffix.

Macro Syntax

Here is a simple macro which defines an AND gate logic block.

```
[[ AND; id; A; B
    id    ; gate=NOT;  a=id --x
    id --x; gate=NAND; a=A;     b=B
]]
```

1. Macros are bracketed using the double-bracket notation: **[[** and **]]**.
2. The macro name (like all symbols) is case sensitive. It appears after the **[[**.
3. Within a macro, statements define new gates and their types, inputs, delays, etc.
4. Macro parameters (by convention) consist of capital letters and are substituted using the same symbol lookup and replacement and concatenation technique described above.

This macro block may be invoked as follows:

```
signal; gate=AND; A=inputA; B=inputB
```

which will result in the creation of the corresponding NAND gate and inverter:

```
signal; gate=NOT; a=signal--x
signal--x; gate=NAND; a=inputA; b=inputB
```

Note that the output of the logic block is called **signal** as one would expect, and that the internal signals generated by the macro have names with suffixes like **–x**.

```
(TBD)
```

Overrides and Redefinition of Gates

Normally gates are defined on a single line such as this:

```
gateName; gate=NAND; a=inputA; b=inputB
```

The first reference to a signal name (such as **gateName**, **inputA** and **inputB**) creates a placeholder internally. Attributes such as the type of gate, inputs and associated delays can be included at the first definition or can be added later. The following are equivalent:

```
gateName; gate=NAND; a=inputA; b=inputB

gateName; gate=NAND
gateName; b=inputB
gateName; a=inputA
```

Any gate attribute can be overridden by simply assigning a new value.

```
gateName; gate=NAND; a=inputA; b=inputB
gateName; a=inputXX
```

Reassignments are important because many circuits consist of groups of elements such as registers or busses that may have common or symmetrically defined elements that are easily described, in bulk, using the interative and symbol-substitution features.

Then particular signals may be overridden to provide unique features for certain elements.

Care must be exercised when using macros in redefinitions since the macro must specify ALL inputs in each case - unlike the native gate example above. Null values will never override a previous definition and can be used to satisfy the “all inputs” rule when using a macro redefinition. Using the AND macro described above these two examples

```
signal; gate=AND; A=inputA; B=inputB
```

and

```
signal; gate=AND; A=inputA; B=
signal; gate=AND; A=; B=inputB
```

yield identical results: an inverter and a single two-input NAND gate with properly defined inputs.

```
(TBD)
```

Gate Maps

(TBD)

Schematic Synthesis

(TBD)

Signal Traces

(TBD)

Stimulus State Machines

(TBD)

Collaboration

Support is provided for storing project designs in the Cloud, and for open-source sharing of designs and implementations.

Every project has a name which is case-sensitive, and may contain letters, digits, **-**, _ and **>**. The source file for any project may be saved to and loaded from the Cloud. Every version saved to the cloud is available. Old versions are permanently preserved and are accessible.

A special syntax is used for accessing project versions as shown here.

example syntax	description
myProj	the most recent version of myProj
myProj(1)	The first saved version of myProj
myProj(12)	The twelfth saved version of myProj
myProj(-1)	The previously saved version of myProj
myProj(-3)	The third previously saved version of myProj

myProj(-4h)	The version that was current four hours ago
myProj(-2d)	The version that was current two days ago
myProj(2017-01-20)	The last version that was created on or before the given date

Whenever a given project is saved, it becomes the current (most recent) version. The actual version number is displayed as part of the name. In addition, the URL for the Simulator is updated with a hash value for the current project version. This allows sharing specific versions among colleagues by simply copying or eMailing the URL.

Cloud Storage

The collaborative features are intended for open-source sharing among multiple workstations or between colleagues. The choice of project names is completely up to the user.

Please be considerate and choose names that are not being used by other developers. Try loading a project by name to see if the name is in use. Your files will not be in danger of being lost since they cannot be overwritten or deleted from the Cloud, and the explicit version numbers will be correct. Multiple projects with the same name will, however, get interspersed version numbers. This will limit the utility of the (-n) “previous version” feature, and the “Current Version” might be the “other” project instead of yours.

There is **no directory** of projects. Do not forget the name that you used. This “feature” allows a certain amount of security-by-obscurity. Give experimental or proprietary designs names that are unlikely to be used or guessed by others.

As a convenience, the Simulator uses a local cookie to preserve the most recently used Project Names on your local machine.

It is expected that there will be team development projects, projects with multiple sub-designs, designs with experimental or temporary files and projects that need to fork and merge. To support these use cases, the project leader should choose a naming convention that will be consistently used by the team members. This can involve creative use of the allowed special characters and the names of team members or groups.

The permanent nature of the Cloud storage and naming of the different versions mean that the project URLs can safely be used as part of the documentation and archives of a project. Current development status may be shared among team members by providing a directory

of URLs with appropriate annotations. How this is implemented is the responsibility of the team leader. I welcome comments and discussion of Best Practices.

Local Files

It is anticipated that the limited source file editing capability provided within the Simulator will be augmented by other tools. Security restrictions prevent browser-based applications such as the Simulator from saving files to the user's computer. This is a major reason for including the Cloud-based Project mechanism.

Saving a source file that has been edited within the Simulator to the local computer is best accomplished using the clipboard Cut and Paste.

The clipboard can also be used to paste source code into the Simulator.

Loading a source file from the local computer can be accomplished using the Choose Local File button. This will use the browser's file selection mechanism to safely read a particular local file.

The Simulator is also a drop target, so the local file system's drag-and-drop feature can be used to drop a source file into the Simulator.

Project Documentation

The Cloud preserves and makes accessible every version of a project file. This provides several unique advantages:

- Audit trail for design and development history,
- Support for permanent documentation,
- Support for regression testing during project evolution,
- Support for Training and Presentation materials
- Support for distributed partitioning of Long Duration or Large System Simulations

Fault Tolerance

Robust system design ensures that all design parameters are kept well within the prescribed operating envelope, and that illegal or fault conditions are handled appropriately within the system.

Modular design using standard, well-characterized elements helps to ensure the behavior of a system on a local scale. Interactions between standard modules and the operation of large-scale features are much harder to characterize. Difficulties include not only the larger number of components but the exponential growth in number of edge and corner conditions that are potential sources of failure.

The Simulator includes a number of features that assist the desinger in building confidence in his design.

Monte Carlo Delay Simulation

Each time a project file is RELOADed the entire gate architecture and State machine environment is recreated based on the selected Project Options. The rebuilding of the gate architecture includes rebuilding the delays of individual signals between gates.

The input and output delay descriptions may be static integer delays in picoSeconds, but they may also include a random variability parameter. During each RELOAD, the variablity element is evaluated and the delay used during this program run will be adjusted accordingly. This allows for introducing a simulation of manufacturing drive and threshold variations, as well as a lumped variation that could involve impedance, temperature, humidity and power supply differences.

Statements in the Simulation State Machines can be used to capture violations of signal timing margins or outright erroneous results. Careful selection of events and totals to log will help to reduce the flood of simulation data to a small set that can be used for design validation.

The Monte Carlo feature allows a particular design to be run an arbitrarily large number of times, each starting with a RELOAD to achieve different, randomly selected, variations in each gate.

In addition to this timing randomization implemented as a feature of the RELOAD operation the Stimulus State Machines can be used to introduce deterministic or random behavior as described in the following sections. The Monte Carlo feature allows selected data to be collected over multiple runs.

Boundary Scan

(TBD)

Noise Injection

(TBD)

Design Guidelines

Several logic design guidelines and conventions are supported or required by the Logic Simulator.

Design for Testability

Design for Test rules require that the entire design be able to be initialized to a known state. This is required for any traditional test method (such as signature analysis) but is especially important for the logic simulator. Every gate must be capable of achieving a known, stable state prior to a meaningful simulation run.

In particular, this means that any feedback loop (such as a gate-delay oscillator or flip-flop) must be able to be held in a known initial state. The simulator uses the signal **POreset#** as an active-low power-on reset signal. The macros that support flip-flops incorporate multi-input NAND gates to clear the flip-flop during the power-on interval.

The special **stable:** conditional expression in the Stimulus State Machine can be used to detect when the entire design has reached a steady-state condition. Usually, this would be used to release the **POreset#** signal to begin the simulation. It may also be used in the simulation of combinatorial logic segments to determine and display propagation delays. In addition, discovering that a design will not stabilize in a reasonable amount of time at power-on is usually an indication that a feedback path does not have a proper reset element.

Symbol Names

The Logic Design Language enforces very few rules with regard to the designer's choice of gate names. The following table lists the names that are explicitly reserved for Simulator internal use.

name	description
0	Logic zero
1	Logic one
POreset#	Power-on reset - active low
gate	Type of gate or Macro name
group	Group ID for Schematic generator
note	Descriptive text for gate
title	Descriptive Title for Simulation Project
a b c d e f g h i	Gate input signal names
[a] [b] [c] [d] [e] [f] [g] [h] [i]	Gate input delays
[o]	Gate output delay

Any printing character may be used in a symbol name. The following table lists characters that have special meaning at certain points in the processing. It is generally wise to avoid the use of reserved characters in symbol names.

character	usage
” “	Double Quotes prevent space compaction in text
[ ]	Square brackets are used for setting signal delay times
=	Equal sign is used for symbolic name assignment
;	Parameter Delimiter
,	Sub-parameter delimiter
@	Gate map Description
/ /	Block Comment Delimiters
// !!	End-of-line Comment Delimiters
[[ ]]	bracket Gate Definition Macros

	Bracket Stimulus State Machine descriptions
== != <> < > <= >=	Relational Operators
?	Wildcard match to **0 1 2 3 4 5 6 7 8 9 A B C D E F**

Title and Note

Descriptive text may be specified by making assignments of the following forms:

```
: title = "Project Title"
: note = "This is my Shift Register"
```

The text is actually HTML, so you can embed tags as desired:

```
: note = "This is <b>my</b> Shift Register"
```

Be sure to properly close your tags to avoid surprising results.

As a convenience, we provide some useful meta-tags for quickly creating colored annotations. These meta-tags are named with one or two capital letters and are magically self-closing as needed.

meta-tag pair	meaning		meta-tag pair	meaning
<R> … </R>	Red text		<RB> … </RB>	Red Background
<G> … </G>	Green text		<GB> … </GB>	Green Background
<B> … </B>	Blue text		<BB> … </BB>	Blue Background
<C> … </C>	Cyan text		<CB> … </CB>	Cyan Background
<M> … </M>	Magenta text		<MB> … </MB>	Magenta Background
<Y> … </Y>	Yellow text		<YB> … </YB>	Yellow Background
<O> … </O>	Orange text		<OB> … </OB>	Orange Background
<W> … </W>	White text		<WB> … </WB>	White Background
<K> … </K>	Black text		<KB> … </KB>	Black Background

Naming Conventions

1. It is recommended that the following characters be used in creating symbol names.

character	description
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z	Upper Case Alphabetic
a b c d e f g h i j k l m n o p q r s t u v w x y z	Lower Case Alphabetic
0 1 2 3 4 5 6 7 8 9	Digits
~ ` ' ! # $ % ^ & * - + . _	Recommended Special Characters

2. To improve clarity in the case of symbol substitution, especially in nested loop contexts, the use of the **&** and **&&** convention may be used.

```
: &subst=XYZ
    : &&inner=AA, BB, CC
        reg &subst &&inner; gate=NAND;     .....
```

In general, however, it is believed that (within reason) removing unnecessary boilerplate (“noise”) characters improves the overall clarity of the design and documentation. Context should indicate which “words” are literal and which will be “filled in” to compose a signal name.

As usual in literate programming the documentation is expected to be predominant. Compact “programming language” features should be clarified by quality explanations.

3. Active-low signal names should end with **#** (pound sign) or **/** (forward slash). I prefer the convention that a signal **sig** passing through an inverter becomes **sig/**, and that signals that are *usually* complementary (such as the Q and Q-bar outputs of a flip-flop) use the **sig** and **sig#** notation. This is how the included macro definitions are structured.

Obviously the real-world and simulated situation involving propagation delays means that the (illgal) **sig** = **sig/** or **sig** = **sig#** states *will* occur, at least momentarily.

Careful use of **sig/** for inverter outputs will cause the automatic consolidation of devices that explicitly invert the same signal. This will reduce the number of generated inverters. Caution should be exercised in the case of inverter chains: signal name elements should be spaced out in the source

```
sig /  ; gate=INV; a=sig
sig / /; gate=INV; a=sig /
```

for clarity and to prevent accidental **//** comments.

4. Signal names should be more than one character long and preferentially begin with a lower-case letter.

5. Use of camelCase is the preferred method of composing names, although the use of _ (underscore) and **-** (dash) may also be appropriate. Choose a style, document it, and then be consistent.

Simulator Configuration

The operation of the Simulator may be configured from within the source file. This allows for repeatable setup for different tests, improves the clarity of collaboarative efforts and can ensure consistent documentation.

Simulator configuration statements take the form of

```
$pragma: option; option...
```

option	description
run	Automatically start running upon load
run-	Do not Automatically start running upon load
sections-	Hide all the Sections of the Simulator Display
map+	Display the Gate Map
map-	Hide the Gate Map

trace+	Show the Trace Display
trace-	Hide the Trace Display
source+	Show the Source Display
source-	Hide the Source Display
schematic+	Show the Schematic Display
schematic-	Hide the Schematic Display
list+	Show the Gate Wire List
list-	Hide the Gate Wire List
log+	Show the Log
log-	Hide the Log
log: size	Set the length of the Log
log: find: text	See the text to find and highlight in the log
log: filter: text	See the text to use to select records in the log
traces: name, name…	Set all traces for the Trace Display
trace: name, name…	Add new trace name traces to the Trace Display
power+	Show the Trace Display cursor in Power Mode
power-	Show the Trace Display cursor in Time Interval Mode
time-	Show the Trace Display cursor in Power Mode
time+	Show the Trace Display cursor in Time Interval Mode
editable+	Show the Source in Editable Mode
editable-	Show the Source in Syntax-Highlight Mode
omit+	Show the Omitted Source Highlighting
omit-	Hide the Omitted Source Highlighting

macro+	Show Source Highlighting for Macros
macro-	Hide Source Highlighting for Macros
machine+	Show Source Highlighting for Stimulus Machines
machine-	Hide Source Highlighting for Stimulus Machines
search: text	Set up the Search field for Gate Wire List
find: text	Set up the Find field for Source Syntax Highlighting
signals: name, name…	Select the components for the Schematic Display
montecarlo: count	Set the number of runs for Monte Carlo simulation
debug+	Enable internal debug mode
debug-	Disable internal debug mode
debug: optionlist	Set internal debug option list
debug: option+	Add new internal debug option to list
debug: option-	Remove internal debug option from list