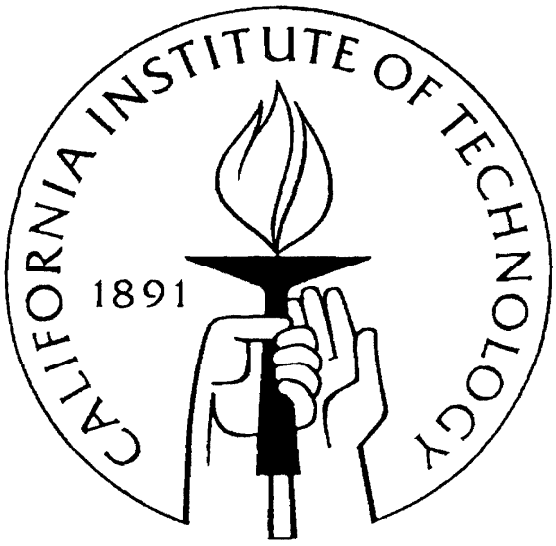


MOSSIM II:
A Switch-Level Simulator for MOS LSI
User's Manual

Randy Bryant
Mike Schuster
Doug Whiting

Computer Science Department
California Institute of Technology

5033:TR:82

CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science

5033:TR:82

MOSSIM II:
A Switch-Level Simulator for MOS LSI
User's Manual

by

Randy Bryant
Mike Schuster
Doug Whiting

This work was funded in part by
Defense Advanced Research Contracts Agency
ARPA Order Number 3771
and by the Caltech Silicon Structures Project

MOSSIM II:
A SWITCH-LEVEL SIMULATOR FOR MOS LSI
USER'S MANUAL

25 January 1983

Randy Bryant
Mike Schuster
Doug Whiting

Table of Contents

List of Figures

1. Introduction

MOSSIM II is a logic simulator based on the switch-level logic model described in the Phd thesis of R. Bryant [2]. It models a MOS digital circuit as a network of nodes connected by transistor "switches" and hence can accurately model such circuit structures as (bidirectional) pass transistors, ratioed and complementary logic, busses, dynamic memory, and charge sharing. Unlike analog circuit simulators, MOSSIM utilizes a logical model and hence operates at speeds comparable to conventional logic gate simulators. Very large designs can be simulated for long input sequences with reasonable computational cost. This program supersedes an earlier version of MOSSIM [1]. It has superior performance, a more general network model, and more powerful simulation capabilities. MOSSIM II is written in Mainsail (TM)* and hence can run on a variety of computer systems.

Besides the basic complementary and ratioed logic circuits allowed by the previous version of MOSSIM, the networks of MOSSIM II can model the effects of charge sharing between nodes of different capacitances as well as model a larger variety of ratioed circuit configurations. Furthermore, the simulator can represent portions of the network as "black boxes" in the form of user-written Mainsail procedures which are dynamically linked into the simulation. Networks can be specified by writing programs in a network description language embedded in Mainsail or can be derived directly from a CIF [5] layout description using a circuit extraction program. A network file can also include calls to other network files so that networks created from a variety of sources can be combined together. Appendix I contains a definition of the network file format so that users can generate networks by whatever means they wish.

The simulation capabilities of MOSSIM II include the ability to define a clocking scheme; to look at or set the state of any node in the network; to set breakpoints based on the network state; and to drive the simulator by user-written Mainsail procedures. MOSSIM II can also apply more stringent tests of a circuit beyond its normal unit delay, switch-level model. It can use ternary simulation [3] to detect potential timing errors. Unlike conventional timing simulators, ternary simulation tests whether the design will operate correctly for all possible circuit delay parameters. Ternary simulation can be augmented to check for potential errors caused by "dynamic charge sharing", i.e. glitches caused by transient charge sharing effects. MOSSIM II can also be run with charge storage disabled (for static circuits), with limits placed on the charge retention time, and with checks for unrestored logic levels (for CMOS).

This manual is divided into sections describing different aspects of the program. Section 2 describes the network model. Section 3 describes the simulation timing models and capabilities. Section 4 discusses details related to the X state. Section 5 documents the simulator commands and usage.

---

*Use of MOSSIM requires a Mainsail run-time license from Xidak, Inc., Sunnyvale, CA

Section 6 describes the network description language embedded in Mainsail. Section 7 describes how networks can be derived by circuit extraction. Section 8 describes now networks can be checked for anomalous and unusual configurations of transistors. Section 9 gives a complete example of the simulator usage. Finally, the appendices describe the various interface formats and other details. Hereafter, the program will be referred to simply as "MOSSIM".

2. Network Model

MOSSIM represents the state of each node with one of three logic values:

0	low
1	high
X	invalid (between 0 and 1), or uninitialized

The additional states seen in other logic simulators (e.g. high impedance) are not required, because such behavior is described by the network model. Two types of nodes are allowed:

Input:	Provide strong signals from sources external to the network (e.g. Vdd, Gnd, clock and data inputs).
Storage:	Have states determined by the operation of the network and can retain these states in the absence of applied signals.

Note that in previous references to the switch-level model, storage nodes were termed "normal nodes". Furthermore, the pullup node of other switch-level simulators (including old MOSSIM) has been made obsolete by a more general transistor model.

Each storage node is assigned a size in the set {1,...,q} to indicate (in a simplified way) its capacitance relative to other nodes with which it may share charge. When a set of connected storage nodes is isolated from any input nodes, they are charged to a logic state dependent only on the state(s) of the largest node(s). Thus the value on a larger node will always override the value on a smaller one. Many networks do not depend on charge sharing for their logical behavior and hence can be simulated with only one node size (q=1). In general, at most two node sizes (q=2) will suffice with high capacitance nodes (e.g. pre-charged busses) assigned size 2 and all other assigned size 1.

Figure 2-1 shows a switch-level model of a three transistor dynamic RAM cell in which the bus node "Bus" is assigned size 2 to indicate that its charge will override the charge on the storage node "Store" during a write operation and will override the charge on the drain node of the storage transistor

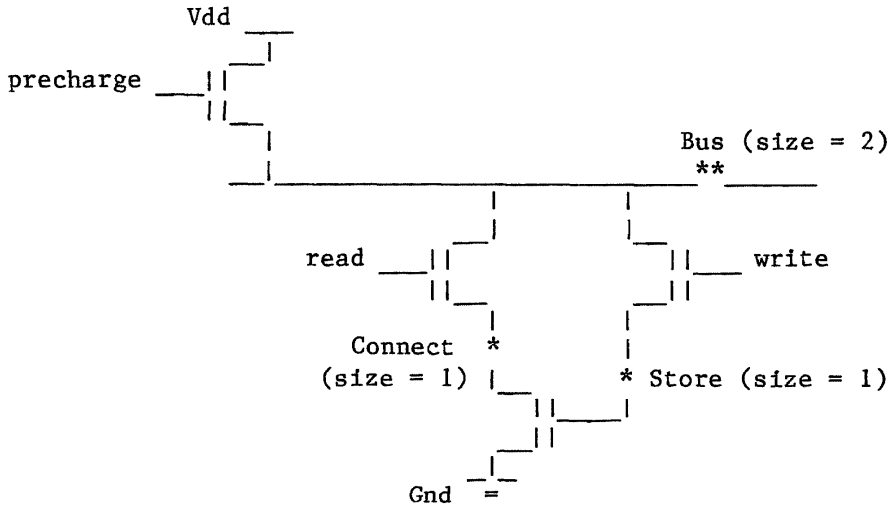
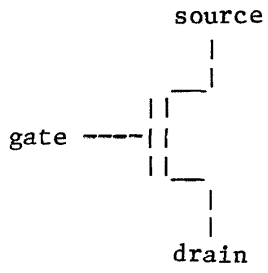


Figure 2-1: Switch-Level Model of Three Transistor Dynamic RAM Cell

"Connect" during a read operation.

Normally, a storage node can retain its state indefinitely in the absence of applied inputs. However, when the static switch = 1, dynamic charge storage is disabled and storage nodes are set to X any time they are isolated from any input nodes. On the other hand, if a limit is placed on the charge hold time (by setting the chargehold limit to a nonzero value), an unrefreshed node state is retained only for the specified number of clock cycles, after which the node is set to X.

A transistor is a three node device as shown below:



No distinction is made between the source and drain connections -- every transistor is a symmetric, bidirectional device. Each transistor has both a strength in the set {1,...,p} and a type in the set **{n,p,d}**. The strength of a transistor indicates (in a simplified way) its conductance when turned on relative to other transistors which may form part of a ratioed path. When

there is at least one path of conducting transistors from a storage node to some input node(s), the node is driven to a logic state dependent only on the strongest path(s), where the strength of a path equals the minimum transistor strength in the path. Thus, a stronger signal will always override a weaker one. Most CMOS circuits do not involve ratioing, and hence can be simulated with one transistor strength (p=1). Most nMOS circuits can be modeled with just two strengths (p=2), with pullup transistors having strength 1 and all others having strength 2. However, circuits involving multiple degrees of ratioing may require more strengths. The simulator will utilize as many node sizes and transistor strengths as are used in the network file with the limitation that $p+q < 16$.

The three transistor types **n**, **p**, and **d** correspond to n-type, p-type, and depletion mode devices, respectively. A transistor acts as a switch between source and drain controlled by the state of its gate node as follows:

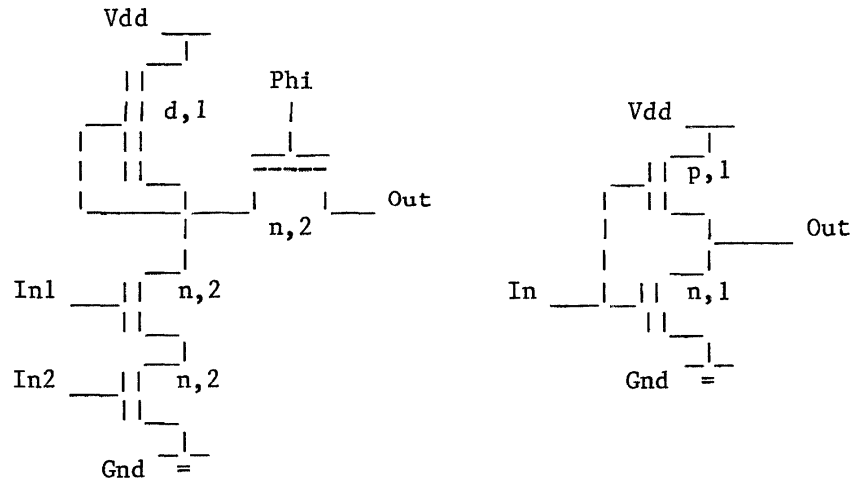
gate	**n**	**p**	**d**
0	open	closed	closed
1	closed	open	closed
X	unknown	unknown	closed

When a transistor is in an "unknown" state it forms a conductance of unknown value between (inclusively) its conductance when "open" (i.e. 0) and when "closed". The simulator models these transistors in such a way that any node with state sensitive to their actual conductances is set to X. Figure 2-2 shows switch-level representations of some simple logic gates.

Normally, a transistor is modeled as conducting both 0 and 1 values without any degradation. In restoring mode, however, (restore switch = 1), a signal with state 1 conducted through an n-type transistor or a signal with state 0 conducted through a p-type transistor will degrade into an X, unless a complementary path is provided. This mode is intended primarily to test for level restoration in CMOS.

The network description may also include functional blocks, i.e. procedures written by the user to provide functional models of parts of the system. The interface for these procedures is described in Appendix II. A more extensive example of a switch-level network is shown in Section 9.

Note that MOSSIM is a logic simulator rather than a circuit simulator and cannot detect many errors in a circuit design. In particular, MOSSIM utilizes a highly simplified model of transistor resistances and node capacitances. As a result it will not always detect improper ratios or marginal cases of charge sharing, even when the network has been derived by circuit extraction. Furthermore, MOSSIM does not detect errors caused by threshold voltage drops (except when the restore switch = 1), and hence yields the same result whether or not you use bootstrapped drivers. As with any tool, MOSSIM should not be used without understanding its limitations.



nMOS Nand gate with pass transistor CMOS inverter

Note: Transistors are labeled with type and strength.

Figure 2-2: Switch-Level Models of Logic Gates

3. Simulation Timing Models

MOSSIM is designed primarily for simulating clocked systems, where a clocking scheme consists of a set of state sequences to be applied cyclically to a set of nodes. It is assumed that the clocks are operated slowly enough in the actual circuit for it to stabilize between each change of clock and input data values. The flow of time is viewed at 3 levels of granularity:

cycle: A complete sequencing of the clocks.

phase: A period in which all clock and input values remain constant.

step: The basic time unit. Within a phase, unit steps are simulated until the network reaches a stable state, or until the step limit is exceeded. Note that this limit can be changed using the Limit command.

For example, a two-phase nonoverlapping clock cycle contains 4 (!) simulation phases to include the periods when both clocks are 0, as is given in the followin c ock declaration:

```
CLOCK Phi1:0100 Phi2:0001
```

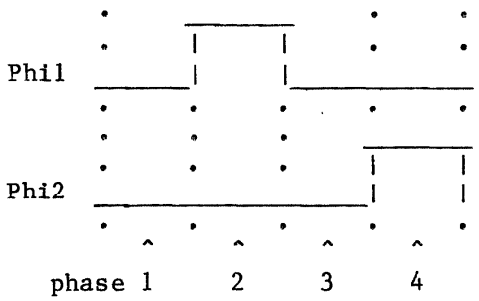


Figure 3-1: Simulation Phases for a Two-Phase, Nonoverlapping Clock

With this declaration, the values shown in Figure 3-1 are applied to nodes Phi1 and Phi2 during each clock cycle.

Normally, a clocking scheme is declared shortly after the network is read in and the simulator is operated at the cycle level. However, the user may monitor the simulation more closely by operating at the phase or step level.

For unclocked systems, the simulator can be run with a null clock (1 phase/cycle, no sequences applied) and, if needed, a combination of functional blocks and breakpoints can be used to change inputs or observe outputs without waiting for the network to stabilize.

Normally, a simple unit delay timing model is used in which a transistor is switched one time unit (i.e. 1 step) after its gate node changes state, but the effects of signals propagating through paths of conducting transistors are simulated with no delay. This timing model only loosely approximates the actual circuit timing model — it is assumed that the circuit's logical behavior is not critically dependent on its timing behavior. A successful simulation in unit delay mode therefore does not show that the circuit is free of timing errors.

The pseudo unit delay timing model (pseudo switch = 1) provides a slight variation on this scheme in which a transistor is switched as soon as its gate node changes state, but simulation events are sequenced in first-in, first-out order, where a switching transistor may cause "events" for both its source and drain nodes. Thus if two events are triggered simultaneously, both will be simulated before any events which they cause are simulated. With this model, the exact ordering of node updating is not totally predictable, but the simulator is less prone to infinite loops caused by matched unit delays (e.g. in cross-coupled Nor gate latches).

The ternary simulation model (ternary switch = 1) provides a rigorous test for possible timing errors in the circuit. That is, if a design simulates successfully in ternary mode, it indicates that the circuit will function correctly for those input sequences regardless of the actual switching and

communication delays in the circuit, providing the clocks run slowly enough. Such a verification is more powerful than can be obtained from even the most accurate circuit or timing simulator -- it is process independent. However, it requires adherence to a somewhat restrictive design methodology, such as a carefully applied two-phase, nonoverlapping clocking scheme, in which no assumptions are made about the internal circuit delays. Note that circuits which rely on the duration of pulses (e.g. clock generators and edge-triggered flip-flops), or on the relative speeds of different feedback paths (e.g. self-timed circuits) will fail ternary simulation.

During ternary simulation, each simulation phase contains two subphases. In the first, all input nodes (including clocks) which change state during this phase are set to X, and the circuit is simulated until a stable state is reached. As a result, all nodes which may be in transition during this phase are set to X. In the second subphase, the input nodes are set to their final values, and the circuit is again simulated until a stable state is reached. As a result, nodes with final states dependent only on the inputs and stable internal nodes will be set to Boolean values (0 or 1), while nodes which depend on the relative times at which transitions occur remain at X.

As an extension of ternary simulation, one can also test for possible instances of dynamic charge sharing (sharing switch = 1) in which a glitch appears on a node due to transient charge sharing effects, and this glitch is captured in some feedback loop. Such glitches are caused by problems of transit delay -- the inability of an established signal path to supply or remove charge fast enough to keep up with newly formed signal path. To perform this test, the simulator operates as during ternary simulation, but in addition sets nodes with potential glitches to X during the first subphase of each ternary phase. Nodes with final states which may be affected by these glitches remain in the X state at the end of the phase. A design which passes this test is immune to all sources of delay in a circuit (for the simulated input sequences): switching delay, communication delay, and transit delay. Note that most static RAM designs will fail this test, because they involve dynamic charge sharing between the storage nodes and the bit lines during a read operation.

4. The X State

MOSSIM assigns the state X to a node any time a unique Boolean (0 or 1) state cannot be assigned, either because that part of the network has not been initialized or because the node voltage potentially lies between the logic thresholds. This state is modeled in a rather pessimistic way -- as if it can represent an arbitrary voltage between 0.0 and Vdd, and such that a transistor with its gate node in the X state can have arbitrary conductance between 0 and its maximum. As a consequence, the X state tends to propagate even in cases where the electrical circuit works correctly. For example, nodes representing state variables which would initialize to random Boolean values during power up in the actual circuit are initialized to X's in the simulator and will remain there until forced into a unique 0 or 1 state by an unambiguous reset sequence. Furthermore, the X state does not satisfy the law of excluded

middle (A + ~A = 1) and hence does not always function as a "don't care" value. For example, a pass transistor multiplexor with an X on one of its control inputs (e.g. the A input of Figure 9-1) will always have an X on its output even if all of the data inputs are identical. At times the user must take special measures to initialize the network at the start of a simulation. This can involve using the Force command to force values onto state nodes, or using the Charge command to set all dynamic storage nodes to Boolean states.

MOSSIM also uses the X state to represent potentially invalid behavior when tests are applied for such design errors as race conditions, dynamic charge sharing, expired dynamic storage, and unrestored logic levels. Once such an error causes an X to appear on a node, the X is propagated in the usual way to model the effect of the error on the rest of the network. When the Explain switch is turned on, the simulator tries to keep track of the causes of the X states and uses special characters to represent these different forms of X. In general, however, the cause of an X cannot be determined with complete reliability, especially when multiple potential causes are present. A node which has never had a Boolean state is marked as U ("uninitialized"). A node driven to both the 1 and the 0 states through equal strength paths is marked S ("short circuit"), and one charged to both 1 and 0 by storage nodes of the same size is marked C ("charge sharing"). A node set to X during the first subphase of ternary simulation is marked T, and which is the origin of a glitch due to dynamic charge sharing is marked D. A node which is set to X because dynamic storage has been disabled (static switch = 1) is marked Z, and one for which the charge storage limit has been exceeded is marked L. All others are marked X.

Users who write Mainsail procedures for functional blocks must simulate the effects of X states on their inputs in a reasonable way, especially if ternary simulation is to be applied. A minimal condition to guarantee that the initial power up will terminate, and that ternary simulation can be applied is for the function block to be monotonic over the partial ordering 0 < X and 1 < X. That is, if some input of a function block equals X, then each output must either equal X or must equal a unique value if a 0, 1, or X were present on the input.

5. Simulator Commands

MOSSIM is invoked interactively by typing the command

```
mossim<cr>
```

while at the monitor level. The program will return with the prompt ">" indicating it is ready to accept user input. In a typical simulation, the user gives commands to read in a network file, to set various parameters such as clock definitions and nodes to be watched, and then to run the network through a sequence of operations. Only enough letters of a command word need be given to disambiguate it from other commands. If the last character of a

command line is a hypen "-", then the next line is treated as a continuation of the current command. The list of possible commands can be displayed by typing "?<cr>".

MOSSIM retains the following information during the simulation:

- The structure of the current network,
- A symbol table indicating the relation between node and vector names and their locations in the network, and constant names and their values,
- The state of the network, i.e. the current logic state of every node and functional block and the current cycle and phase numbers, and
- The state of the runtime switches and limits, the clock definitions, the names of the nodes to be watched, the breakpoints, and the names of the source and destination files.

5.1. Vectors

A vector is a symbolic name representing an ordered set of nodes. Vectors can be defined either in the network file or with the Vector command. Vectors provide a convenient means of setting or retrieving the state of a set of logically related nodes. A vector of size 1 serves as an alias for the original node name.

5.2. Constants

A constant is a symbolic name representing an ordered set of logic values. Its definition is declared with the Constant command. Constants provide a convenient means of setting nodes or vectors to constant values.

5.3. Commands

Commands can be classified according to the nature of their actions:

System Set Up:					
	READ	WRITE	CLOCK	WATCH	UNWATCH
Symbol Table Manipulation:					
	VECTOR	CONSTANT	PREFIX	UNPREFIX	
Run Control:					
	INITIALIZE	DUMP	LOAD	UPDATE	
	CYCLE	PHASE	STEP		
Node Manipulation:					
	GET	VERIFY	STATUS		
	SET	FORCE	UNFORCE	CHARGE	
Breakpoint Manipulation:					
	BREAK	UNBREAK			
Operational Control:					
	SOURCE	EXECUTE	COPY	COMMENT	
	SWITCH	LIMIT	HELP	QUIT	EXIT

The commands are discussed in the following sections. The syntax of a command and its arguments are given in an informal BNF. All arguments must be separated by blanks or tabs. Non-terminals are underlined; { } denotes repetition any number of times including zero; [ ] indicate optional factors; | denotes "or"; and ( ) denote grouping. Upper and lower case distinctions are ignored, i.e. xyz denotes the same name as Xyz, XYZ, etc. The following definitions apply:

filename =	<any valid file name>
modname =	<any valid module name>
node =	name
vector =	name
constant =	name
block =	name
value =	{ state } | { octalstate } | { hexstate } | node | vector | constant
state =	0 | 1 | X
explainstate =	U | C | S | T | D | L | Z
octalstate =	0 | 1 | ... | 7 | X
hexstate =	0 | 1 | ... | 9 | A | ... | F | X
format =	b | o | h
time =	s | p | c
name =	<any combination of characters except blank>
number =	<a positive integer>

Whenever a filename with no file extension is given as a command argument, a default extension will be appended to it. If the file associated with the resulting filename cannot be opened, an attempt will be made to open the file

associated with the unextended argument filename. If this open fails an error message will be returned. The default extensions are command dependent and are given in the following descriptions.

Constant state values may be specified in binary, octal, or hexidecimal format. On output, the octal and hexidecimal digit X is used when one or more of the corresponding state digits is X. On input, an X will set all corresponding state digits to X. A /format is given as a command argument to resolve the ambiguity between binary, octal, and hexidecimal values.

5.4. System Set Up

5.4.1. READ

read filename

A network file in the NTK format is read, a new symbol table is produced, and all node states are set to X, except for the nodes Vdd and Gnd, which are set to 1 and 0, respectively. The default file extension is "ntk".

5.4.2. WRITE

write filename

A network file in the NTK format is written, including the definitions of all vectors. The default file extension is "ntk".

5.4.3. CLOCK

clock { node:value }

This command defines a clocking scheme consisting of a set of nodes and a set of state sequences to be applied to these nodes. Any previously defined clocking scheme will be removed. Thus, all nodes and state sequences involved in a clocking scheme MUST be specified on one command line.

During the simulation, the program will manipulate the clock nodes according to the declaration. The sequences must all be the same (nonzero) length. If the command is given with no arguments, then a null clock will be used (1 phase per cycle, no sequences applied).

5.4.4. WATCH

watch { node | vector | block | /number | /* | /format }

This command allows the user to declare a set of nodes, vectors, and functional blocks whose states are to be printed out on every clock cycle after a particular clock phase. The state of a functional block is obtained by invoking the block's Status procedure with the null string as an argument. This procedure must write appropriate internal state information to the terminal.

Normally, the states are reported after the current phase. However, if /n appears in the sequence, where n is a number, then any succeeding node, vector, or functional block name in the command line (until the next /n or /*) will be reported after phase n. If /* appears in the sequence, then any succeeding node, vector, or functional block name in the command line (until the next /n) will be reported after every phase. A clocking scheme should be declared with the Clock command before the Watch command is specified.

If a /format appears in the sequence, then any succeeding nodes and vectors in the command line (until the next /format) will be reported in the specified binary, octal, or hexidecimal format.

Example:

```
watch sum carry /2 sum load /* refresh
```

5.4.5. UNWATCH

unwatch { node | vector | block | /number | /* | * }

This command reverses the effect of the Watch command. If * appears in the sequence, then all current node, vector, and functional block watches for the specified phase(s) are removed.

5.5. Symbol Table Manipulation

5.5.1. VECTOR

vector [/format] name { node | vector }

This command declares that the vector named name will refer to the ordered

set of nodes and vectors given. A vector of length 1 acts as a synonym for the node. An optional /format may be given to specify the vector's default input/output format.

Examples:

```
vector sum sum[3] sum[2] sum[1] sum[0]
vector /h carry&sum carry[3] sum
vector bit4 alu.1/slice.1/cell.4/a
```

5.5.2. CONSTANT

constant [/format] name { value | /format }

This command declares that the constant named name will refer to the ordered set of values given. Constants are useful for generating mnemonics for often used values. An optional /format may be given to specify the constant's default input/output format. If a /format appears in the sequence, then any succeeding values in the command line (until the next /format) are assumed to be in the specified binary, octal, or hexidecimal format.

Examples:

```
constant 4zeros 0000
constant 8zeros 4zeros 4zeros
constant /h decimal20 14
```

5.5.3. PREFIX

prefix { name | name:name | ? }

This command allows the user to specify a "working prefix" and "prefix macros" that are used to modify all argument node, vector, and constant names in subsequent commands. The working prefix is specified via an argument of the form name, and name/ is simply prefixed to all subsequent argument names, unless the first character of the name is "/". If the command is given with no arguments, the last /text in the current working prefix will be removed.

Prefix macros are specified via arguments of the form prefix:macro and result in the use of macro/text in place of the argument prefix/text given in subsequent commands.

If ? appears as an argument, the current working prefix and prefix macros are listed.

For example, if the command

```
prefix cell.1 1:register.bit1 2:register.bit2 3:/cell.2
```

is given, then names in the second column will be used in place of names in the first column:

/carry	carry
carry	cell.1/carry
1/carry	cell.1/register.bit1/carry
2/carry	cell.1/register.bit2/carry
3/carry	cell.2/carry
/2/carry	2/carry

5.5.4. UNPREFIX

```
unprefix { name | * }
```

This command deletes the specified prefix macros. If * appears in the sequence, then all prefix macros are deleted and the working prefix is set to null. If the command is given with no arguments, then the working prefix will be set to null.

5.6. Run Control

5.6.1. INITIALIZE

```
initialize
```

This command resets all node states to X, except for the nodes Vdd and Gnd, which are set to 1 and 0, respectively. The clock is reset as well.

5.6.2. DUMP

```
dump filename
```

This command saves the current network state in the specified file. Only stable networks may be dumped. In addition to the network state, the following information is saved: cycle and phase numbers, clock definitions,

the static and restore switches, and the chargehold limit. Pending future sets, verifies and breakpoints are not saved. The default file extension is "dmp".

5.6.3. LOAD

load filename

This command loads the network state from a file created by the Dump command. In addition to the network state, the following information is restored: cycle and phase numbers, clock definitions, the static and restore switches, and the chargehold limit. Loading clears any pending future sets, but not verifies or breakpoints. The default file extension is "dmp".

5.6.4. UPDATE

update { modname | * }

This command allows the user to replace existing functional blocks in the network with recompiled versions without having to reread the entire network. The replacement is performed for all instances of a specified functional block module (modname) or for all instances of all functional blocks in the network (*).

The replacement procedure for each instance of a functional block is as follows. First, the current state of the existing functional block is obtained by invoking its Dumpstate procedure. The new version is then linked in, and its Instantiate and Initialize procedures are invoked. Finally, the Loadstate procedure is invoked to restore the previously saved state. Of course, formats used by the Dumpstate and Loadstate procedures in the existing and new versions must be compatible.

5.6.5. CYCLE

cycle [number]

This command causes the network to run for the specified number of cycles. If no argument is given, 1 is assumed. If the network starts mid-cycle, the current cycle is completed and then n- cycles are simulated, where n is the argument to the command.

5.6.6. PHASE

phase [number | * | .5]

This command causes the network to run for a number of phases. If n is given, where n is a number, then n phases are simulated. However, if the network starts mid-phase, the current phase is completed and then n-1 phases are simulated. If no argument is given, 1 is assumed. If * is given, one cycle is simulated, unless the network starts mid-cycle, in which case the current cycle is completed. If .5 is given and the ternary switch = 1, one subphase of a ternary phase is completed, unless the network starts in mid-subphase, in which case the current subphase is completed. If .5 is given and the ternary switch = 0, 1 is assumed.

5.6.7. STEP

step [number | * | *.5]

This command causes the network to run for a number of steps. If n is given, where n is a number, then n steps are simulated. If no argument is given, 1 is assumed. If * is given, one phase is simulated, unless the network starts mid-phase, in which case the current phase is completed. If *.5 is given and the ternary switch = 1, one subphase of a ternary phase is completed, unless the network starts in mid-subphase, in which case the current subphase is completed. If *.5 is given and the ternary switch = 0, then * is assumed. The * and *.5 options are useful for watching nodes after each step.

5.7. Node Manipulation

5.7.1. GET

get { value | block | /format }

This command returns the values of the specified nodes, vectors, constants, and functional blocks. The value of a functional block is obtained by invoking the block's Status procedure with the null string as an argument. This procedure must write appropriate internal state information to the terminal.

Each output line starts with the cycle, phase and step number. If a /format appears in the sequence, then any succeeding values in the command line (until the next /format) will be reported in the specified binary, octal, or

hexidecimal format. If no arguments are given, then the states of the nodes, vectors, and functional blocks being watched on the current clock phase will be returned.

5.7.2. VERIFY

verify { node:value | vector:value | /number | /format }

This command checks the state of the nodes and vectors listed. If the state of a node or vector is not equal to the specified value, its current state is listed. Normally the checking is performed immediately. However, if /n appears in the sequence, where n is a number, then any succeeding node or vector names in the command line (until the next /n) will be queued as "future" verifies for phase n. Future verifies for phase n are checked just after phase n is simulated. If a /format appears in the sequence, then any succeeding values in the command line (until the next /format) are assumed to be in the specified binary, octal, or hexidecimal format.

5.7.3. STATUS

status { node | vector | block <any text string> |
? | ?state | ?explainstate }

This command returns status information about the specified nodes and vectors. With this command, the user can perform local network tracing and debugging. Status information for a node includes the node's type, size, state; the r, u, and d values as described in Bryant's thesis [2] (kappa and gamma are abbreviated k and g, respectively); and the transistors in the node's fanout and connectivity sets. (A transistor is in its gate node's fanout set and in both its source and drain node's connectivity sets). When the redundant switch = 1, the simulator temporarily removes transistors from a node's fanout set when it can be determined that their switching will not change the network state. As a consequence, the Status command does not give completely accurate information: the fanout list includes only non-redundant transistors, and the r, u, and d values may be incorrect. More accurate information can be obtained by turning this switch off and running the simulator for at least one step. For efficiency reasons, however, the switch should be turned on whenever possible.

If block appears in the sequence, the Status procedure of the functional block with block as its instance name is invoked with the remainder of the command line passed as an argument. This procedure is usually used to write appropriate internal state information to the terminal. The argument may be used to control what information is written.

If ? appears in the sequence, a list of all perturbed nodes is returned

(i.e. those nodes which are scheduled to be simulated.) This option helps in the debugging of network oscillations. If ?state appears in the sequence, a list of all nodes in the specified state is returned. If ?explainstate appears in the sequence, and the explain switch = 1, a list of all X nodes with the specified explanation is returned.

5.7.4. SET

set { node:value | vector:value | /number | /format }

This command sets the nodes and vectors to the specified values. One may set the state of any node in the network, but unless the node is an input node, its state may subsequently be altered by the network activity. To prevent this from occurring, the Force command should be used. Normally the states are set immediately. However, if /n appears in the sequence, where n is a number, then any succeeding node or vector names in the command line (until the next /n) will be queued as "future" sets for phase n. Future sets for phase n are applied just before phase n is simulated. If a /format appears in the sequence, then any succeeding values in the command line (until the next /format) are assumed to be in the specified binary, octal, or hexidecimal format.

Example:

```
set sum:4zeros /2 load:1
set /h sum:F
```

5.7.5. FORCE

force { node:value | vector:value | /number | /format | ? }

Forcing a node temporarily turns it into an input node which can drive its signal onto other nodes. The state of a forced node cannot be altered by the operations of the simulator. Only forcing or setting it to some other value will change it. A node should be taken out of forcing mode (by unforcing it or setting it) when normal operation is to be resumed. This feature is useful for isolating a section of the network and testing it independently from other parts. It also helps in initializing feedback loops. This command has the same syntax as the Set command. If ? appears in the sequence, a list of all currently forced nodes is returned.

5.7.6. UNFORCE

unforce { node | vector | * }

This command takes the listed nodes and vectors out of forcing mode without changing their logic states. If * appears in the sequence, then all currently forced nodes are unforced.

5.7.7. CHARGE

charge { state:state }

This command locates all charged nodes in the network (i.e. those isolated from any input nodes) whose current state is o and sets them to a the new state n, where o:n appears in the argument sequence. This command is useful for initializing that part of the network state represented by stored charge to either 0 or 1.

5.8. Breakpoint Manipulation

5.8.1. BREAK

break { /time | node:value | vector:value | /format | ? }
time = s | p | c

This command defines breakpoints by associating "breakpoint" values with specified nodes or vectors. Whenever the state of the node or vector becomes equal to its breakpoint value during simulation, the breakpoint causes an interrupt which terminates the Step, Phase or Cycle command. Normally, breakpoints cause interrupts at the completion of the unit step in which the node's or vector's state becomes equal to its breakpoint value. However, if /time appears in the sequence, where time is s, p or c, then any succeeding breakpoints in the command line (until the next /time) will cause interrupts only at the completion of a step, phase or cycle, respectively. A node or vector has at most one breakpoint value at any given time, any existing breakpoint values are forgotten when a new one is associated with the node or vector. If a /format appears in the sequence, then any succeeding values in the command line (until the next /format) are assumed to be in the specified binary, octal, or hexidecimal format. If ? appears in the sequence, a list of all breakpoints is returned.

5.8.2. UNBREAK

unbreak { node | vector | * }

This command deletes breakpoints associated with the listed nodes and vectors. If * appears in the sequence, then all breakpoints are deleted.

5.9. Operational Control

5.9.1. SOURCE

source filename

This command declares that the simulator input is to come from the named file rather than the terminal until all commands in that file have been executed. Thus the user can construct a file containing an arbitrary sequence of commands and run them through in batch mode. For example, a sequence of system set up and symbol table manipulation commands can be put into a file to avoid retyping them every time the simulation is run. Command files may contain nested source commands. The default file extension is "src".

5.9.2. EXECUTE

execute modname <any text string>

This command causes the named Mainsail module to be linked in and transfers control to it with the remainder of the command line passed as an argument. This module can then drive the simulator through a set of interface procedures and upon exit return control to the simulator. This command allows the user to customize the simulator interface according to the particular application or even develop a new front end. The interface between the simulator and a driver module is defined in Appendix III.

5.9.3. COPY

copy filename

With this command, the user can "wallpaper" the simulation output so that everything which is printed on the terminal is also written into the specified file. This provides a hard copy record of the simulation run. If no argument

is given, any previous wallpapering is turned off. The default file extension is "cpy".

5.9.4. COMMENT

```
comment <any text string>
```

This command causes the comment text to be inserted into the output stream. This aids the documentation of the simulation run.

5.9.5. SWITCH

```
switch { switchname:0 | switchname:1 | ? }
switchname = ternary | sharing | static | restored | pseudo |
             redundant | explain | statistics | echo
```

This command turns the named runtime switches on (1) or off (0). If ? appears in the sequence, the current status of all switches is returned. The initial values and meanings of the switches are as follows:

ternary	0	Check for races using ternary simulation
sharing	0	Check for dynamic charge sharing
static	0	Disable dynamic charge storage
restored	0	Check for unrestored CMOS logic levels
pseudo	0	Utilize pseudo-unit delay model
redundant	1	Ignore effects of redundant transistors
explain	0	Explain cause of unknown states
statistics	0	Print out statistics while running
echo	1	Print commands read from source files

When the ternary switch is on, the ternary simulation model described in Section 3 will be applied. If the sharing switch is also on, then the checks for dynamic charge sharing will be applied. This switch has no effect unless the ternary switch is on.

When the static switch is on, dynamic charge storage is disabled as described in Section 2.

When the restored switch is on, any nodes with degraded signals (1 threshold above Gnd or below Vdd) are set to X. This mode is used primarily when testing CMOS circuits.

The pseudo switch enables the pseudo unit delay timing model as defined in Section 3.

When the redundant switch is on, the simulator ignores transistors which cannot affect the network state by switching. It is normally turned on, although the simulator may automatically turn it off and then back on for certain operations. Occasionally, the user may wish to turn it off manually to get more useful status information. This will slow down the simulation somewhat.

When the explain switch is on, the simulator makes an attempt to diagnose the cause of an X state on a node. Note that this explanation is not always reliable. The following character codes are used in place of X when the node's state is printed:

X	cause unknown
U	uninitialized node
C	charge sharing error
S	short circuit
T	ternary delay sensitive
D	dynamic charge sharing error
L	chargehold limit error
Z	static storage error

When the statistics switch is turned on, the simulator will print some run time statistics after each clock cycle, including the number of unit steps simulated, the amount of CPU time required, and various obscure details. Hackers may find this information interesting, and it also provides some indication of the amount of activity being simulated.

Finally, the echo switch indicates whether commands read from a source file are to be copied to the output stream before being executed. It can be shut off to reduce the amount of print out during long simulations.

5.9.6. LIMIT

```
limit { limitname:number | ? }
limitname = step | chargehold | error | pagewidth
```

This command sets the named runtime limits to the specified numbers. If ? appears in the sequence, the current status of all limits is returned. The initial values and meanings of the limits are as follows:

step	100	Maximum number of steps per simulation phase
chargehold	0	Charge retention time limit (in cycles)
error	500	Number of error reports before aborting simulation
pagewidth	80	Width of page for printout (in characters)

The step limit determines the maximum number of unit steps that will be taken during each simulation phase. The default limit of 100 was chosen as an upper bound for most designs based on past experience. Some networks may require a higher limit, however, particularly unclocked systems. A simulation may also exceed this limit when the network oscillates due to matched unit delays, in which case the simulation should be continued in pseudo unit delay mode (pseudo switch = 1).

The chargehold limit determines how long (in clock cycles) a node can retain a charged state. Setting the limit to 0 (the default) means that charge can be retained indefinitely. If you really want a time limit of 0, i.e. to disable dynamic storage, you should set the static switch to 1.

The error limit determines the number of error messages which will be tolerated before the entire simulation is aborted. This limit should be set to a low value for batch runs.

The pagewidth limit is used to format output lines.

5.9.7. HELP

help [<command> | ?]

This command returns information about a command. If ? appears as an argument, a list of all commands is returned.

5.9.8. QUIT

quit

This command causes MOSSIM to be terminated.

5.9.9. EXIT

exit

Like Quit, this command causes MOSSIM to be terminated.

6. Network Description Language

By embedding the network description language (NDL) in Mainsail, the user is given the full power of a programming language for describing a network to the simulator. However, the user must have some familiarity with the programming language and compiler to fully utilize these capabilities, and much of the NDL syntax was dictated by what could be embedded in Mainsail. NDL is actually defined by a set of macro definitions which convert the program into the Mainsail code for a module when the Mainsail compiler is run, along with a set of Mainsail modules. NDL allows a network to be defined as a hierarchy of nets, where each net can contain commands to create nodes, transistors, and function blocks, as well as calls to other nets. A net can also include calls to other network files so that subnetworks created by other means (e.g. by circuit extraction) can be incorporated into the network. The nodes in a network created by an NDL program obey a hierarchical naming convention based on the net hierarchy.

The file "ndl.mi" (all files and programs mentioned in this paragraph are to be found in the same directory as MOSSIM) must be included as a SOURCEFILE in the NDL code. Furthermore, net definitions for simple inverters, nor gates, and nand gates can be obtained from either the nMOS library, "nmoslb.mi", or the CMOS library, "cmoslb.mi". After the NDL description is compiled, the user should invoke the program NDL (by typing NDL<cr> at the monitor level), This program will prompt the user for the module name and the network file name. The module will be executed to expand the network and create the network file. Any error messages will be reported both to the terminal and to the output file.

6.1. Nodes

Each node is named according to the point within the hierarchy of net calls in which it is created. That is, if a node is created in a net, then its name in the actual network will be "prefix/nodename", where "nodename" is its locally declared name, and "prefix" is the current prefix string reflecting the path in the hierarchy from the top to this particular net call. The user controls how this prefix is constructed according to how a net is called. Ordinarily, when a net is called, the prefix string becomes "oldprefix/netname". However a net can be called with an instance name as its last argument to give a prefix "oldprefix/netname.instancename" to provide unique names for nodes with otherwise identical calling sequences. The instance name can be an arbitrary string as long as it contains no embedded blanks. Furthermore, a net can be called with the keyword "IgnoreP" as its last argument to make the new prefix equal to the old prefix. The only requirement of the name prefixing is that nodes created at different points in the network hierarchy have unique names in the network file. To use either an instance name or IgnoreP in a net call, the net must be defined with the keyword "iname" as its last parameter. Examples of some node names created by the NDL program of Figure 9-2 are reg/storebar, and2.1/middle, and2.2/middle, and and2.1/nand/middle[1].

Nodes can be grouped into vectors which are actually represented by Mainsail

arrays and hence are indexed by integer subscripts. A vector is constructed (allocated) either from a set of existing nodes by the Makevec command, in which case the indices range from 1 up to the size of the vector, or by creating a new set of nodes with the Inptvec, Storagevec, Smallvec, or Largevec commands, in which case the resulting network nodes will have names of the form "prefix/vectorname[index]". The command Keepvec when called with a vector as argument will put the declaration of a vector named "prefix/vectorname" into the network file consisting of the current nodes in the vector. The command KeepvecRev acts similarly but with the nodes in reverse order (highest index first).

A net is defined by code of the following general form:

```
Net(netname(paraml, ..., paramk ; iname));

        < local node, vector, namevector, and variable declarations >

BeginNet

        < main body >

EndNet;
```

A net definition is actually just a 'sugared' Mainsail procedure and hence the parameter list, local declarations, and main body obey the usual Mainsail conventions.

The Include command is used to instantiate a subnetwork described by a MOSSIM network file. The parameters to this command include a vector of nodes and a vector of strings to specify a mapping between nodes in the current network and node names in the network file so that the combined networks will be wired together properly. Even the power and ground connections must be specified explicitly. The declarations of these nodes in the included network file will be overridden. The names of all other nodes in the network file will be prefixed by a string of the form "prefix/filename.instancename/", "prefix/filename/", or "prefix/", depending on whether the final argument of the Include command is "instancename", "", or IgnoreP, respectively, where "prefix" is the current prefix string. The user must ensure that the resulting node names do not coincide with the names of any other nodes in the network. The inclusion of a subnetwork into the main network does not actually occur until the network is read in by the simulator. Thus, the user must make sure the network file for the subnetwork is available at simulation time, at which time any error checking will be applied. The default file extension for a network file is "NTK". The Include command requires an argument of data type NameVector, which is simply an array of strings.

Large networks should be broken up into a number of modules to allow separate compilation. For each module, the programmer should create a module interface file containing declarations of all nets to be accessed from outside the module. These declarations are of the form

```
iNet(netname(paraml, ..., paramk ; iname));
```

and should appear just like the declaration in the actual file except for the keyword "iNet" instead of the usual "Net". Any module which calls this module should read in the interface file with a SOURCEFILE command. The nMOS library module with source code in the file nmoslb.m and interface specification in the file nmoslb.mi provides an example of the use of modules in NDL.

The following subsections describe the individual NDL constructs. The user is encouraged to study the examples of Figures 9-2 and II-2, and to print out some network files to learn more about NDL.

6.2. Node Primitives

Node	Declare a node. A node is actually represented in Mainsail by a pointer to a data structure. Thus each node must first be declared, and then allocated with the Inpt, Storage, Small, or Large statement. The programmer must control the prefixing in such a way that no two nodes have the same name in the network.
Vector	Declare a vector. A vector is actually represented by an array of nodes. Thus each node must first be declared, and then allocated with the Inptvec, Storagevec, Smallvec, Largevec, or Makevec statement.
Inpt	Allocate an input node. The format is inpt(name), where name has been declared previously. The input nodes Vdd and Gnd are already declared and allocated globally.
Storage	Allocate a storage node. The format is storage(name,size), where name has been declared previously and size is a positive integer.
Small	Allocate a storage node of size 1. The format is small(name).
Large	Allocate a storage node of size 2. The format is large(name).
Makevec	Allocate a vector consisting of a set of already allocated nodes. The format is makevec(vecName, [node1, ... , nodek]).
Inptvec	Allocate a set input nodes and allocate a vector containing these nodes. The format is inptvec(vecName, LowerBound, UpperBound).
Storagevec	Allocate a set of storage nodes and allocate a vector containing these nodes. The format is storagevec(vecName, nodeSize, LowerBound, UpperBound).

Smallvec — Allocate a set of storage nodes of size 1 and allocate a vector containing these nodes. The format is storagevec(vecName, LowerBound, UpperBound).

Largevec — Allocate a set of storage nodes of size 2 and allocate a vector containing these nodes. The format is storagevec(vecName, LowerBound, UpperBound).

Appendvec — Append one or more nodes onto an already allocated vector. The format is either appendvec(name, node), or appendvec(vecName, [node1, ... , nodek]).

Keepvec — Save the association between a vector name and the vector's current set of nodes in the network listing file. The format is keepvec(vecName).

KeepvecRev — Same as Keepvec, except that the list of nodes is reversed in the vector definition.

6.3. Transistor Primitives

Ntrans — Create an n-type transistor. The format is ntrans(strength, gateNode, sourceNode, drainNode). There are two NDL-supplied defaults for strength: weak=1, strong=2. As an example, the pulldown of an inverter might be ntrans(strong,in,out,gnd).

Ptrans — Same as ntrans, but makes a p-type transistor.

Dtrans — Same as ntrans, but makes a d-type transistor.

6.4. Net Definitions

Net — Start the definition of an NDL net. The format is Net(mname), where mname is the text for a Mainsail procedure declaration. This primitive begins the procedure definition, but also does the necessary bookkeeping. Any procedure qualifier must precede Net; for example, BOOLEAN Net(route(VECTOR in,out)). All local variable declarations (including nodes, vectors, and namevectors) must follow the Net declaration but precede the BeginNet.

iName — Used as the last parameter of a Net to allow for optional instance names and ignoring prefixes.

ignoreP — Passed as the last parameter to a call of a net declared using iName to prevent appending to the current prefix by the call.

BeginNet — Marks the beginning of the actual procedure body of the net.

EndNet — Marks the end of the procedure body of the net.

iNet — Used to declare NDL nets as interface procedures of a module. The format is <procedure-qualifiers> iNet(name), where mname is just as for Net.

Comment — Write the argument string to the network listing file for documentation purposes. The format is Comment(string). The argument string cannot contain may embedded semicolons (;).

6.5. Block Calls

Block — Declare an external module as a function block. The format is:

```
Block("modName","argument",invec,outvec,"instance").
```

ModName is the name of the Mainsail module encoding the function block. The argument string may be used to pass arbitrary information to the block. For example, the name of a personalization file could be the argument to a generic PLA module. The argument string cannot contain any embedded double quotes ("). Invec and outvec are vectors of input and output nodes. The optional instance argument is used to form an instance name for the block of the form "prefix/modName.instanceName", "prefix/modName" for arguments "instanceName", and "", respectively. The argument IgnoreP is not allowed here. Each function block must have a unique instance name in the network file.

6.6. Subnetwork Inclusion

Include — Declare a network file to be included as a subnetwork. The format is:

```
Include("FileName", nodeVector, nameVector,"instance")
```

The argument "Filename" is the name of the netw k file describing the subnetwork. The nodeVector and nameVector arguments specify a mapping between nodes in the current network and the names of nodes in the subnetwork file. The nodeVector argument is of type Vector (i.e an array of nodes), and the nameVector argument is of type Namevector (i.e. an array of strings). The optional instance argument may be

either "instancename", "", or IgnoreP.

Namevector — Declare a name vector. A name vector is actually represented by an array of strings. Thus, a name vector must first be declared and then allocated by the MakeNameVec command.

MakeNameVec — Allocate a name vector consisting of a set of strings. The format is MakeNameVec(vecName, [string1, ... , stringk]).

AppendNameVec — Append a set of strings onto an existing name vector. The format is AppendNameVec(vecName, string) or AppendNameVec(vecName, [string1, ... , stringk]).

7. The CONVERT Program

The program CONVERT provides a way to generate a network file in the NTK format from a circuit description produced by the circuit extraction program EXTRACT of T. Hedges [4]. It can also be used to do limited editting of a network file.

The extractor traces the electrical connectivity in an MOS layout, finds the transistors, and generates a file specifying the electrical circuit in terms of the connectivity, the areas of the different layers for each node (from which the node capacitance can be computed), and the type, length and width of each transistor (from which the resistance can be computed.) To convert such a circuit into a switch-level network, the nodes must be assigned types (input or storage) and discrete sizes based on their relative capacitances. Furthermore, the transistors must be assigned discrete strengths based on their relative resistances. The CONVERT program applies a set of heuristics to assign sizes and strengths, but in some cases the user must override these with explicit assignments. Furthermore, before the heuristics can be applied, the types of the nodes must be assigned either in the extraction file, or with the Type or Size command.

CONVERT is invoked interactively by typing the command

convert<cr>

while at the monitor level. The program will return with the prompt ">" indicating it is ready to accept user input. Only enough letters of a command word need be given to disambiguate it from other commands. The list of possible commands can be displayed by typing "?<cr>".

7.1.Commands

The commands are discussed in the following sections. The following definitions apply:

filename =	<any valid file name>
node =	name
size =	number
transistor =	$number
strength =	number
gate =	name
source =	name
drain =	name
name =	<any combination of characters except blank>
number =	<a positive integer>
real =	<a positive real number>

A transistor can be referred to by a physical name of the form $number that specifies an actual location in the network. The STATUS command may be used to obtain the physical names of all transistors connected to a specified node.

7.1.1. EXTRACT

extract filename

A network file generated by the EXTRACT program is read. The default file extension is "xtr". All electrical parameters must be specified via the Parameter command before the network is read.

7.1.2. NETWORK

network filename

A network file in the NTK format is read in. The default file extension is "ntk".

7.1.3. STRENGTH

strength { nmos | cmos | ratio:real [,real] | transistor:strength }

This command allows the user to assign transistors strengths either globally

throughout the network (nmos, cmos, or ratio:real,real) or on an individual basis (transistor:strength). The global strength assignment should not be attempted until all input nodes have been specified via the Type command.

If the option nmos is specified, all depletion transistors with an input node as drain will be assigned strength 1, all other transistors are assigned strength 2.

If the option cmos is specified, all transistors will be assigned strength 1.

If the option ratio:real,real is specified, a simple cluster analysis is performed on groups of transistors that may form ratioed paths. The transistors in each group are first sorted by resistance and then partitioned into clusters such that transistors in the same cluster differ in resistance by no more than the factor k2 and transistors in different clusters differ in resistance by at least the factor k1, where the argument ratio:k1,k2 was specified. The transistors in each cluster are assigned the same strength.

The values of k1 and k2 should be somewhat less than the ratio of the size of an inverter's pullup resistor to the size of its pulldown resistor to account for the extractor's miscalculation of the sizes of non-orthogonal transistors. Typically, a value between 3.0 and 4.0 is used for both k1 and k2.

If the first factor specified is zero, then all transistors will be assigned the strength 1. If the second factor is omitted, then it is assumed to be equal to the first.

7.1.4. SIZE

```
size { ratio:real [ ,real ] | node:( input | size ) }
```

This command allows the user to assign node sizes either globally throughout the network (ratio:real,real) or on an individual basis (node:input or node:size). The global size assignment should not be attempted until all input nodes have been specified via the Type command.

The technique used for global size assignment performs a simple cluster analysis on groups of nodes that may share charge. The nodes in each group are first sorted by capacitance and then partitioned into clusters such that nodes in the same cluster differ in capacitance by no more than the factor k2 and nodes in different clusters differ in capacitance by at least the factor k1, where the argument ratio:k1,k2 was specified. The nodes in each cluster are assigned the same size.

A typical value for both k1 and k2 is at least 3.0.

If the first factor specified is zero, then all nodes will be assigned the

size 1. If the second factor is omitted, then it is assumed to be equal to the first.

7.1.5. TYPE

type { node:(input | size) | transistor:(n | p | d) }

This command allows the user to assign node or transistor types on an individual basis. A node may be given the type input (node:input) or storage by specifying a size (node:size). A transistor may be made an n, p, or d-type device.

7.1.6. NODE

node { /g | /s | /d | transistor:node | node:node }

This command allows the user to change the gate, source, and drain node of transistors on an individual basis (transistor:node) or globally throughout the network (node:node). The global update locates all transistors in the network whose gate, source, or drain node is old and changes that node to new, where old:new appears as an argument. The arguments /g, /s, and /d are used to specify which of the gate, source, or drain nodes are to be changed, respectively (until the next /g, /s, or /d in the command line). One of /g, /s, or /d must be specified before transistor or node arguments are given.

7.1.7. INSERT

insert (input | size) { name } |
(n | p | d) strength gate source drain

This command inserts a node or a transistor into the network. A node may be given the type input or storage by specifying the argument input or size, respectively. A node may also be given a set of node names. A transistor is specified by a type (n, p, or d), a strength, and the names of its gate, source, and drain nodes.

7.1.8. DELETE

delete { node | transistor }

This command deletes the specified nodes and transistors from the network. A node may not be deleted if it is connected to any transistors or functional blocks.

7.1.9. STATUS

status { node | transistor }

This command returns status information about nodes and transistors. With this command, the user can perform local network tracing and individual strength and size assignments. Status information for a node includes the node's type, size, capacitance (in pico-farads), x and y coordinates (in microns) and the transistors in the node's fanout and connectivity sets. Status information for a transistor includes the transistor's physical name, type, strength, and the names of its gate, source, and drain nodes, resistance (in kilo-ohms), and x and y coordinates (in microns).

7.1.10. PARAMETER

parameter { /r | /c | param:real | ? }
param = diffusion | polysilicon | metal | n-type | p-type | d-type

This command sets the named electrical parameters of the layers and transistors to the specified real values. Either /r or /c must be specified before any values are given. Values appearing after /r and /c (until the next /r or /c) are assumed to be resistances in kilo-ohms per square and capacitances in pico-farads per square micron with respect to the underlying substrate, respectively. If ? appears in the sequence, the current values of all the electrical parameters are listed.

7.1.11. WRITE

write filename

A network file in the NTK format is written. Any node whose size is as yet unassigned is given size = 1. Any transistor whose strength is as yet unassigned is given strength = 1. The default file extension is "ntk".

7.1.12. HELP

help [<command> | ?]

This command returns information about a command. If ? appears as an argument, a list of all commands is returned.

7.1.13. QUIT

quit

This command causes CONVERT to be terminated.

7.1.14. EXIT

exit

Like Quit, this command causes CONVERT to be terminated.

8. The MOSCHK Program

The program MOSCHK performs a static analysis of a network looking for anomalous and unusual configurations of transistors. Note that only a simple set of heuristics is used, thus, errors may be missed and false diagnostics may be reported.

MOSCHK begins the analysis by reporting various network statistics. The total number of input and storage nodes, n-type, p-type and d-type transistors, and functional blocks is reported.

A set of transistor checks is then performed. A diagnostic is generated for each transistor whose gate, source, and drain node satisfy any of the following conditions:

source = Gnd and drain = Vdd
gate = Vdd
gate = Gnd
source = drain
(gate = source or gate = drain) and (n-type or p-type)

An attempt is made to classify each transistor according to its use as either a pullup, pulldown, or pass transistor. The total number of transistors in each class is reported. A diagnostic is generated for unusual cases such as a p-type pulldown transistor, or a d-type pass transistor. A transistor is flagged as unused if there is no path from either of its source and drain to transistor gates or functional block inputs.

A similar set of node checks is then performed. An attempt is made to classify each node according to its use along pullup, pulldown or both pullup and pulldown paths. The total number of nodes in each class is reported. A diagnostic is generated for a node if there is no path from the node to a transistor gate or a functional block input, i.e., if the node is unused and has no fanout. A diagnostic is also generated for a storage node if there is no path from the node to an input node or a functional block output. A check is then performed in an attempt to detect multiple threshold drop errors. Finally, any node that can never be set to 0 and/or 1 is reported.

MOSCHK is invoked interactively by typing the command

moschk<cr>

while at the monitor level. The program will return with the prompt ">" indicating it is ready to accept user input. Only enough letters of a command word need to be typed to disambiguate it from other commands. The list of possible commands can be displayed by typing "?<cr>".

8.1.Commands

The commands are discussed in the following sections. The following definition applies:

filename = <any valid file name>

8.1.1. READ

read filename

A network file in the NTK format is read. The default file extension is "ntk".

8.1.2. TECHNOLOGY

technology (nmos | cmos)

This command allows the user to define the network's fabrication process technology. The technology must be specified before the Check command is used.

8.1.3. CHECK

check [filename]

This command performs the static analysis of the network. A summary file is generated with the extension "sum", and an error report file is generated with the extension "chk". If no file name is specified, both the summary and the error report will be listed on the terminal. The process technology must be specified via the Technology command before the analysis is performed.

8.1.4. HELP

help [<command> | ?]

A brief summary of the various check performed by MOSCHK is returned. In addition, if a command work appears as an argument, information about the command will be returned. If ? appears as an argument, a list of all commands is returned.

8.1.5. QUIT

quit

This command causes MOSCHK to be terminated.

8.1.6. EXIT

exit

Like Quit, this command causes Moschk to be terminated.

9. Simulation Example

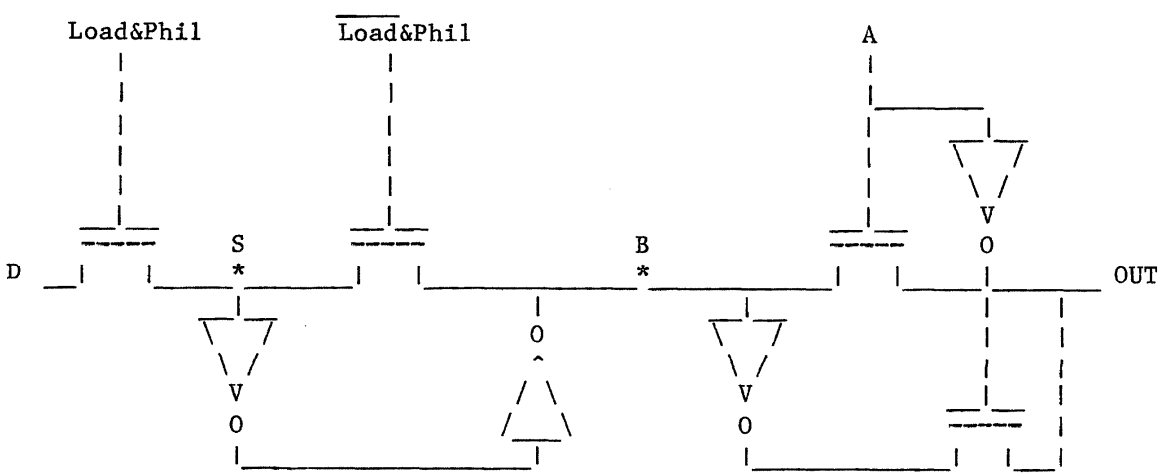


Figure 9-1: Quasi-Static Register with Multiplexor on Output

The following example shows how the simulator is operated and how ternary simulation can be used to detect potential timing errors. Figure 9-1 shows a quasi-static register (i.e. static while Phil=1) which drives a multiplexor with the intention of computing the function A=B. The NDL source program for this circuit is shown in Figure 9-2.

```
begin "quasi"
sourcefile "ndl.mi"; #files are on 'mossim' directory
sourcefile "nmoslb.mi";

     Net(reg(node in, out, store, cload, cloadbar; iname));
          node storebar;
     beginNet
          small(storebar);
          ntrans(strong, cload, in, store);
          inv(store, storebar); inv(storebar, out);
          ntrans(strong, cloadbar, out, store);
     endNet;

     Net(and2(node in1, in2, out; iname)); # two input And gate
          node outBar; vector ins;
     beginNet
          small(outBar);
          makevec(ins, [in1, in2]);
          nand(ins, outBar);
          inv(outBar, out);
     endNet;

     Net(equiv(node a, b, out; iname)); # multiplexor
          node abar, bbar;
     beginNet
          small(abar); small(bbar);
          inv(a, abar); inv(b, bbar);
          ntrans(strong, a, b, out);
          ntrans(strong, abar, bbar, out);
     endNet;

Net(chip(iName)); # the complete network
     node phi1, load, D, S, B, A, OUT, loadbar, cload, cloadbar;
beginNet
     inpt(phi1); inpt(load); inpt(D); inpt(A);
     small(S); small(OUT); small(B);
     small(loadbar); small(cload); small(cloadbar);
     inv(load, loadbar); #form the control signals
     and2(phi1, loadbar, cloadbar, "1"); #instance name required
     and2(phi1, load, cload, "2");
     reg(D, B, S, cload, cloadbar); #assemble the elements
     equiv(A, B, OUT);
endNet;

initial procedure;
     chip(ignoreP);        #call chip, ignore common prefix 'chip/'

end "quasi";
```

Figure 9-2: NDL program for the Network of Figure 9-1

The following text documents a simulation run in unit delay mode.

```
>comment Simulation in unit delay mode
>read quasi.ntk
    19 nodes, 24 transistors, 0 blocks
>source quasi.src
>clock phi1:010
>watch /1 D S B A OUT /2 load D S B A OUT
>set load:1 D:1 A:0
>cycle
  1.1| D:1 S:X B:X A:0 OUT:X
  1.2| load:1 D:1 S:1 B:1 A:0 OUT:0
>set load:0
>cycle
  2.1| D:1 S:1 B:1 A:0 OUT:0
  2.2| load:0 D:1 S:1 B:1 A:0 OUT:0
>comment Try changing A on different clock phases
>set A:1
>cycle
  3.1| D:1 S:1 B:1 A:1 OUT:1
  3.2| load:0 D:1 S:1 B:1 A:1 OUT:1
>set A:0
>cycle
  4.1| D:1 S:1 B:1 A:0 OUT:0
  4.2| load:0 D:1 S:1 B:1 A:0 OUT:0
>set /2 A:1
>cycle
  5.1| D:1 S:1 B:1 A:0 OUT:0
  5.2| load:0 D:1 S:1 B:1 A:1 OUT:1
>set /2 A:0
>cycle
  6.1| D:1 S:1 B:1 A:1 OUT:1
  6.2| load:0 D:1 S:1 B:1 A:0 OUT:0
>quit
```

In this example, the commands to set up the simulation were read from a file named "quasi.src" using the SOURCE command. This file contained the following commands:

```
clock phi1:010
watch /1 D S B A OUT /2 load D S B A OUT
```

The clock has been defined so that a SET command without phase specification will change the node while Phi1 is a stable 0. The WATCH command specifies nodes to be watched on both phase 1 and phase 2 of each cycle. Each output line of the simulation run starts with the cycle and phase number.

The unit delay simulation indicates correct functionality regardless of when input A is changed with respect to Phi1.

Now let us try applying ternary simulation to the same design.

```
>comment Simulation in ternary mode
>switch ternary:1
>read quasi.ntk
    19 nodes, 24 transistors, 0 blocks
>source quasi.src
>clock phi1:010
>watch /1 D S B A OUT /2 load D S B A OUT
>set load:1 D:1 A:0
>cycle
  1.1| D:1 S:X B:X A:0 OUT:X
  1.2| load:1 D:1 S:1 B:1 A:0 OUT:0
>set load:0
>cycle
  2.1| D:1 S:1 B:1 A:0 OUT:0
  2.2| load:0 D:1 S:1 B:1 A:0 OUT:0
>comment Dump state here in anticipation of troubles ahead
>dump quasi.dmp
>comment Try changing A on different clock phases
>set A:1
>cycle
  3.1| D:1 S:1 B:1 A:1 OUT:1
  3.2| load:0 D:1 S:1 B:1 A:1 OUT:1
>set A:0
>cycle
  4.1| D:1 S:1 B:1 A:0 OUT:0
  4.2| load:0 D:1 S:1 B:1 A:0 OUT:0
>set /2 A:1
>cycle
  5.1| D:1 S:1 B:1 A:0 OUT:0
  5.2| load:0 D:1 S:X B:X A:1 OUT:X
>comment Reload the state and try changing A from 1 to 0 on phase 2
>load quasi.dmp
>set A:1
>cycle
  3.1| D:1 S:1 B:1 A:1 OUT:1
  3.2| load:0 D:1 S:1 B:1 A:1 OUT:1
>set /2 A:0
>cycle
  4.1| D:1 S:1 B:1 A:1 OUT:1
  4.2| load:0 D:1 S:X B:X A:0 OUT:X
>quit
```

This example shows that as long as A is changed while Phi1 is a stable 0, no error occurs, because the register feedback is turned off. However, if A is changed while Phi1 is 1 (or is in transition), a sneak path forms from the output of the inverter driven by B through both pass transistors of the multiplexor, giving a transient value on B which could potentially be latched into the register. Therefore both B and S are set to X. This sneak path seems quite plausible when A changes from 0 to 1, (cycle 5) because there will be a period while A equals 1, but the inverter driven by A still has an output of

1. On the other hand, for a sneak path to form as A changes from 1 to 0 (cycle 7), the inverter driven by A must switch to 1 and the pass transistor it controls must turn on before the pass transistor controlled by A turns off. This example shows that ternary simulation uses a very pessimistic delay model: delays of arbitrary length can occur in any wire or transistor in the circuit.

This example also demonstrates that ternary simulation only detects errors for the particular test values and timings (with respect to the clocks) that are simulated. In general the worst cases occur when as many inputs as possible change simultaneously.

I. The NTK Network Description Format

Users who wish to interface their own network generation programs to the simulator may generate network description files in the NTK format. The syntax of the NTK format is given in an informal BNF. All terminals must be separated by blanks, tabs, end-of-lines (eols), or end-of-pages (eops). Non-terminals are underlined; { } denotes repetition any number of times including zero; [ ] indicate optional factors; | denotes "or"; and ( ) denotes grouping. Upper and lower case distinctions are ignored, i.e. xyz denotes the same name as Xyz, XYZ, etc.

```
network =       { [ comment ] cell } body
cell =          header body
header =        c cellname { parameter } ;
cellname =      name
parameter =     name
body =          { node | equate | forget | transistor |
                block | instance | vector |
                include | comment } end
node =          ( i | s size ) { name } ; [ { attribute } ; ]
size =          number
attribute =     /name value
value =         name
equate =        e name { name } ;
forget =        f { name } ;
transistor =    ( n | p | d ) strength gate source drain ;
                [ { attribute } ; ]
strength =      number
gate =          name
source =        name
drain =         name
block =         b module argument blockname
                { input } ; { output } ;
module =        name
argument =      " { name } "
blockname =     name
input =         name
```

```
output =          name
instance =        h cellname instancename { parameter } ;
instancename =    name | . | -
vector =          v vectorname name { name } ;
vectorname =      name
include =         m filename instancename { name includename } ;
filename =        name
includename =     name
comment =         | { name } ;
end =             .
number =          < any positive integer >
name =            < any combination of characters, except blank,
                    tab, eol, and eop; may not start with #>
                  | #number
```

The overall network file consists of a set of subnetwork definitions followed by the main body.

The cell statement allows the user to define the structure of a subnetwork and later instantiate the subnetwork a number of times with different nodes as actual parameters. A cell definition consists of a header statement which defines the name of the cell and its formal parameters, followed by the body of the cell. Each formal parameter represents a node that can be referred to within the cell body. Formal parameters will be replaced with actual nodes when the cell is instantiated. All nodes referenced within the cell body must be either formal parameters or declared "local" within the cell body with the node statement.

The node statement declares an input or storage node. Each storage node is given an integer size. Nodes may optionally be assigned one or more names. All names within a cell and the main body must be unique. A node may be subsequently referred too either by one of its names or by a physical name of the form #number. The n formal parameters of a cell are assigned physical names #1, #2, ..., #n, and the local nodes have physical names #n+1, #n+2, etc. The nodes in the main body are assigned physical names #1, #2, etc. The use of physical node names in subsequent statements reduces the number symbol table lookup operations required and significantly decreases the time spent reading the network file.

The equate statement adds new names to a previously declared node.

The forget statement deletes names from a previously declared node. Deleted names may be subsequently redefined and referenced, but this is not recommended.

The transistor statement declares a transistor of a specified type along with an integer strength and gate, source, and drain nodes.

Node and transistor statements may optionally contain a property list of attributes. An attribute consists of an arbitrary name of the form /name followed by an arbitrary value. The attributes currently recognized for nodes

are

/x	x coordinate (microns)
/y	y coordinate (microns)
/c	lumped capacitance (pico-farads)

The attributes currently recognized for transistors are

/x	x coordinate (microns)
/y	y coordinate (microns)
/r	resistance (kilo-ohms)

The instance statement instantiates a cell with a set of actual parameter nodes. The names of all local nodes within the cell are prefixed by a string of the form "prefix/cellname.instancename/", "prefix/cellname/", of "prefix/", depending on whether the instancename argument is "instancename", ".", or "-", respectively, where "prefix" is the current prefix string. The resulting node names must not be equal to the names of any other nodes in the network.

The block statement declares an instance of a block with the specified module name, argument string, block name, and list of input and output nodes. The block name and current prefix string are used to generate a unique block instance name of the form "prefix/block name". The argument string cannot contain any embedded double quotes (").

The vector statement declares that the argument vector name will refer to the ordered set of nodes given. The vector name and current prefix string are used to generate a unique vector instance name of the form "prefix/vector name".

The include statement is used to instantiate a subnetwork described in another NTK format network file. The parameter to this statement is a list of pairs of nodes and strings that specify a mapping between nodes in the current network and node names in the included network so that the combined networks will be wired together properly. Even the power and ground connections must be specified explicitly. The declarations of these nodes in the included network file will be overridden. The names of all other nodes in the included network file will be prefixed by a string of the form "prefix/filename.instancename/", "prefix/filename/", or "prefix/", depending on whether the instancename argument is "instancename", ".", or "-", respectively, where "prefix" is the current prefix. The resulting node names must not be equal to the names of any other nodes in the network.

The comment statement may be used for arbitrary annotation or documentation. The statement is terminated by a semicolon (;).

II. Functional Block Interface

It is often convenient to be able to simulate parts of a system as functional blocks, that is procedural objects which compute functions from a set of inputs to a set of outputs, possibly involving internal state. For example, if some part of the system has not yet been designed at the transistor level, the user may prefer to specify its operation procedurally. Similarly, if the design is to interface with other components such as memory chips, this external environment can best be described at a functional level.

To combine switch-level and procedural representations, the interface between objects represented differently must be clearly defined. MOSSIM achieves this by requiring that the system satisfy the following structural and behavioral rules:

- an input to a functional block must have no effects on the node supplying it with a logic value,
- all output nodes of functional blocks must be type Storage,
- no two outputs of functional blocks may be wired together,
- transistors connected to an output node of a functional block must have no effect on the logic value of that node, and
- the outputs of a functional block must be monotonic functions of the block's inputs over the partial ordering $0 < X$ and $1 < X$.

Thus the interactions between the different representations can be described in terms of the logic values 0, 1, and X, and information will flow in a single direction at each connection point.

Functional blocks are implemented as user-written Mainsail module of type "blockclass". The code for such a module should include a SOURCEFILE command to read the file "block.mi" from the MOSSIM directory. This file contains the interface declarations for the following user specified procedures:

```
PROCEDURE instantiate
  (STRING argument, instancename; INTEGER inputsize, outputsize);
```

This procedure is invoked by the Read command when the network is read. The argument string and instance name from the network file is passed along with the number of input and output nodes.

```
PROCEDURE initialize;
```

This procedure is invoked by the Initialize command. All internal state variables should be set to their initial values, such as X.

```
PROCEDURE evaluate(INTEGER ARRAY(1 to *) inputs, outputs);
```

This procedure is invoked the the Step, Phase, and Cycle commands ONLY WHEN one or more input nodes of the block change value. The procedure should compute any new output node values and update any internal state variables appropriately. All output node values MUST be returned in the argument output array WHETHER OR NOT they changed value. The logic state values 0, 1, and X are encoded in the argument arrays as the integers 0, 1, and 2, and for convenience, the identifiers "lo", "hi", and "x" have been defined for them in the interface.

```
PROCEDURE dumpstate(POINTER(textfile) statefile);
```

This procedure is invoked by the Dump command when the network state is saved. The current values of any internal state variables should be written to the argument textfile.

```
PROCEDURE loadstate(POINTER(textfile) statefile);
```

This procedure is invoked by the Load command when the network state is restored. The values of any internal state variables should be read from the argument textfile.

```
PROCEDURE status(STRING argument);
```

This procedure is invoked by the Status command when the instance name of the block is specified with the remainder of the command line passed as an argument. This procedure is usually used to write appropriate internal state information to the terminal. The argument may be used to control what information is written. Note that this procedure should NOT change the values of any internal state variables.

As an example, Figure II-1 shows a procedural implementation of a quasi-static register intended to model the register shown in Figure 9-1. Table lookup techniques are used whenever possible to implement ternary functions for generality and efficiency. Notice that the sneak paths found in the example shown in Figure 9-1 would not be found with this implementation since the output node of a functional block is not affected by connecting transistors. Figure II-2 shows an NDL source program that uses this procedural implementation as a component in a shift register circuit.

```
BEGIN "qreg"
  SOURCEFILE "block.mi";
  MODULE(blockclass) qreg;
  # the instance name and state variable
  STRING instance; INTEGER s;
  # a mapping from load, in, and s to next s
  INTEGER ARRAY(lo TO x, lo TO x, lo TO x) supdate;

  PROCEDURE initialize; s := x;

  PROCEDURE dumpstate(POINTER(textfile) statefile);
  # write state as 0, 1, or X, unparsestate defined in interface
  cwrite(statefile, unparsestate(s));

  PROCEDURE loadstate(POINTER(textfile) statefile);
  # read 0, 1, or X, parsestate defined in interface
  s := parsestate(cread(statefile));

  PROCEDURE status(string argument);
  # write current state of s to terminal
  TTYWRITE("qreg ", instance, " s:", cvcs(unparsestate(s)), eol);

  PROCEDURE instantiate
    (STRING argument, instancename; INTEGER inputsize, outputsize);
  BEGIN instance := instancename;
       IF inputsize NEQ 3 OR outputsize NEQ 1 THEN
            TTYWRITE("Invalid arguments to qreg ", instance, eol);
       NEW(supdate); INIT supdate
       (lo, hi, x,   # load lo, input lo
        lo, hi, x,   # load lo, input hi
        lo, hi, x,   # load lo, input x
        lo, lo, lo, # load hi, input lo
        hi, hi, hi, # load hi, input hi
        x,  x,  x,   # load hi, input x
        lo, x,  x,   # load x, input lo
        x,  hi, x,   # load x, input hi
        x,  x,  x);  # load x, input x
  END;

  PROCEDURE evaluate(INTEGER ARRAY(1 TO *) inputs, outputs);
  BEGIN
       # input and output variables, dont use loadbar=[inputs[3]]
       DEFINE in=[inputs[1]],load=[inputs[2]],out=[outputs[1]];
       # assume load and loadbar are nonoverlapping
       out := s := supdate[load, in, s];
  END;
END "qreg";
```

Figure II-1: Procedural Implementation of a Quasi-Static Register

```
BEGIN "qshift"

SOURCEFILE "ndl.mi";

        Net(chip(integer n; iname));
                node phi1, phi2, in;
                vector outs, blockins, blockouts;
                integer i;
        beginNet
                inpt(phi1); inpt(phi2); inpt(in); #define input nodes
                smallvec(outs, 0, n); # make output vector
                outs[0] := in; # insert input node
                keepvec(outs); # keep vector definition
                FOR i := 1 UPTO n DO
                BEGIN
                  IF i MOD 2 = 1
                    THEN makevec(blockins, [outs[i-1], phi1, phi2])
                    ELSE makevec(blockins, [outs[i-1], phi2, phi1]);
                  makevec(blockouts, [outs[i]]);
                  block("qreg", "", blockins, blockouts, cvs(i));
                END;
        endNet;

initial procedure;
begin
        integer n;
        ttywrite("enter length of shift register: ");
        n := cvi(ttyread);
        chip(n, ignoreP);
end;
END "qshift"
```

Figure II-2: NDL Shift Register program using the functional block

III. Simulator Driver Interface

The first argument of an EXECUTE command is the name of a driver module of type "driverclass". An example of the code for a driver module is shown in Figure III-1. The code for a driver module should include a SOURCEFILE command to read the file "driver.mi" from the MOSSIM directory. This file contains interface declarations for both the simulator and the driver module. A driver module contains a procedure:

```
PROCEDURE execute(POINTER(simclass) simulator; STRING argline);
```

The simulator calls this procedure and passes it a pointer back to the simulator along with the remainder of the command line. The simulator is then driven through a set of interface procedures and, upon exit, control returns to the simulator. The three most important interface procedures are declared as follows:

```
BOOLEAN PROCEDURE command(STRING cmdline; OPTIONAL BOOLEAN echo;
                          PRODUCES OPTIONAL BOOLEAN broken);

STRING PROCEDURE get(STRING name);

PROCEDURE print(STRING line);
```

The procedure 'command' causes the simulator to execute the string 'cmdline' as if it had been given to the simulator directly. It returns TRUE if some error occurred during execution. If a TRUE value is given for the argument 'echo', the command line will be inserted into the output stream before being executed. The argument 'broken' is return true if a breakpoint occurred during the execution of the command.

The procedure 'get' returns the state of the node, vector, or constant given as argument. It returns the null string if the name is not found. The procedure 'print' causes the argument to be inserted into the output stream followed by an end of line.

Additional interface procedures are given to parse the argument line, and to convert between integer values and state vectors. They are documented in the interface declaration file "driver.mi" on the MOSSIM directory.

Figure III-1 shows a procedure which will enumerate all Boolean values of a vector and simulate one cycle for each value. This procedure can be used to create a truth table for a combinational circuit.

```
# make truth table by enumerating all values of a vector
# and simulating one clock cycle for each value

BEGIN "enum"
# Read in declaration file. File is on the MOSSIM directory.
  SOURCEFILE "driver.mi";

  MODULE(driverclass) enum;

  PROCEDURE execute(POINTER(simclass) simulator; STRING argline);
  BEGIN
       STRING name, setting;
       INTEGER m, i;
       name := simulator.gettoken(argline); #get name of vector
       m := length(simulator.get(name));    # find out how long it is
       FOR i := 0 UPTO (2**m - 1) DO # enumerate all values
       BEGIN
            #convert i to state vector
            #makesetting procedure defined in interface
            setting := simulator.makesetting(i, m);
            #set the vector
            simulator.command("set " & name & ":" & setting);
            #simulate one cycle
            simulator.command("cycle");
       END;
  END;

END "enum";
```

Figure III-1: Driver Procedure to Build Truth Table

IV. Installation

The source and documentation files of MOSSIM and its supporting programs may be found on the distribution tape. The tape contains the following files (star (*) is the filename wildcard character):

rel.src	A list of Module names
*.m	Mainsail Module files
*.mi	Mainsail Module Interface files
*.lmk	Mainsail Librarian command files
*.doc	Documentation files

All of the files on the tape must be read into a single directory, hereafter referred to as the "MOSSIM directory".

The files "sysmod.mi" and "sysmod.m" must be edited to incorporate system dependent file names and procedures. The variable "defaultdirectory" in "sysmod.mi" must be equated to the name of the MOSSIM directory. The system dependent procedure "gettime" in "sysmod.m" which returns elapsed CPU time must be implemented. Versions of "gettime" for the DEC-20, UNIX VAX, and VMS VAX are available by setting one of the variables "isdec20", "isvaxunix", or "isvaxvms" in "sysmod.mi" to 1, respectively. If this procedure cannot be easily implemented for other systems, the variable "isgeneral" can be set to 1, causing "gettime" to return the constant 0. This will only affect the information returned by MOSSIM's statistics switch.

The procedures "openfile", "adddefaultextension", and "noextension" in "sysmod.m" are used to append extension names of the form ".name" to argument file names. These procedures may have to be modified to conform to the host system's file naming conventions.

Once "sysmod.mi" and "sysmod.m" are updated, the Mainsail modules comprising the MOSSIM programs may be compiled. This is accomplished by running the Mainsail Executive and executing the Mainsail Compiler COMPIL with the following commands where <cr> means carriage return:

```
*compil,
>cmdfile rel.src
><cr>
```

After all of the modules are compiled (and possibly assembled), the Module Library files containing the object modules may be created by executing the Mainsail Module Librarian MODLIB with the following commands:

```
*modlib
MODLIB: read mossim.lmk
MODLIB: read convert.lmk
MODLIB: read moschk.lmk
MODLIB: read ndl.lmk
MODLIB: <cr>
```

On some systems, the default number of pages reserved for module storage in library files is insufficient, and MODLIB will return a library full error message. In this case, the user must specify a larger number of pages to reserve as a argument to the "create" command in the appropriate *.lmk file. Please refer to the Module Librarian section of the Mainsail Utilities User's Guide for assistance.

Finally, the executable bootstrap files for the programs MOSSIM, CONVERT, MOSCHK and NDL may be created by executing the Mainsail Configurator CONF once for each program with the following commands where the name of the MOSSIM directory is substituted in place of the string <MOSSIM directory>:

```
*conf
CONF: bootfilename mossim.mac
CONF: commandstring
mossim,
preloadlibrary <MOSSIM directory>mossim.lib
<cr>
CONF: <cr>

*conf
CONF: bootfilename convert.mac
CONF: commandstring
convrt,
preloadlibrary <MOSSIM directory>convert.lib
<cr>
CONF: <cr>

*conf
CONF: bootfilename moschk.mac
CONF: commandstring
moschk,
preloadlibrary <MOSSIM directory>moschk.lib
<cr>
CONF: <cr>

*conf
CONF: bootfilename ndl.mac
CONF: commandstring
dondl,
preloadlibrary <MOSSIM directory>ndl.lib
<cr>
CONF: <cr>
```

The resulting bootstrap files "mossim.mac", "convert.mac", "moschk.mac", and "ndl.mac" may then be assembled and/or compiled using the appropriate system dependent utilities. The user should refer to the host system's Mainsail User's Guide for assistance. The resulting executable files may then be copied to the host system's public directory.

REFERENCES

[1]
Bryant, R.
MOSSIM: A Logic Level Simulator for MOS-LSI.
Technical Report 80-21, MIT VLSI Memo, July, 1980.

[2]
Bryant, R.
A Switch-Level Simulation Model for Integrated Logic Circuits.
PhD thesis, Massachusetts Institute of Technology, 1981.

[3]
Brzozowski, J. A., and M. Yoeli.
On a Ternary Model of Gate Networks.
IEEE Transactions on Computers :178-183, March, 1979.

[4]
Hedges, T.
CIF Extractor Help File.
SSP Program Documentation.

[5]
Mead, C., and Conway, L.
Introduction to VLSI Systems.
Addison Wesley, 1980.