SimMOS

by **Martin Newell**

Version of December 5, 1980 12:27 PM

[Ivy]<DA>SimMOS>SimMOS.bravo,.press

PREFACE

SimMOS is a mixed mode switch level and functional simulator for n-channel MOS circuits. No timing information is represented in the simulator, although sequential circuits, and even some asynchronous circuits, can be simulated. Parts of a circuit can be replaced with user-written procedures that model the behaviour of those parts. This permits the simulation of critically timing dependent subcircuits, enhances efficiency, and permits top down development of circuits by starting with only their functional simulation, and substituting implementations of parts of the circuit as they are developed.

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road
Palo Alto   California 94304

Contents

1. Introduction

SimMOS is a mixed mode switch level and functional simulator for n-channel MOS circuits. No timing information is represented in the simulator, although sequential circuits, and even some asynchronous circuits, can be simulated. Parts of a circuit can be replaced with user-written procedures that model the behaviour of those parts. This permits the simulation of critically timing dependent subcircuits, enhances efficiency, and permits top down development of circuits by starting with only their functional simulation, and substituting implementations of parts of the circuit as they are developed. However, use of this technique is not yet documented.

1.1 Background

SimMOS is based on the switch level simulator MOSSIM from MIT. MOSSIM was originally written by Randy Bryant to support the VLSI course given at MIT by Lynn Conway in the Fall of 1978. Since then it has been further developed by Randy Bryant and has been adapted by several people into various versions in several languages. A version by Chris Terman, written in Lisp, was used to diagnose the design of the Scheme79 microprocessor. It is this version that formed the basis for the Mesa implementation, SimMOS.

1.2 Acknowledgements

To Randy Bryant and Chris Terman at MIT, and to Alan Bell, for several long discussions and help in sorting out the basic algorithms. Voluminous feedback from the first heavy user, Jim Cherry, was instrumental in getting SimMOS into a usable state.

2. Operation

SimMOS is implemented as a set of programs in Mesa. Several versions are available for running on different configurations of machines, ranging from a single bank Alto version, through an XM Alto version using high memory for data storage, to D0 and Dorado versions that make efficient use of high memory. The D version on the Dorado is about 60 times faster than the XM Alto version. Only the XM Alto and D machine versions are distributed.

While the basic SimMOS functions are implemented in Mesa, the user interface is provided by running SimMOS under the interactive programming system JaM [JaM]. JaM provides a simple interpreter that supports the JaM programming language, and from which Mesa programs can be called. This programmibility of the user interface proves to be invaluable for developing test procedures for circuits being simulated. (It also provides the basic mechanism by which functional simulation can be implemented along with the switch level simulation). For the purposes of this document the term SimMOS will be used to cover the combination of the basic implementation and the JaM user interface.

2.1 Input Formats

Circuits can be specified to SimMOS in one of two formats: the .sim format or as a JaM program, or as a combination of both. The .sim format is provided to allow input from Clark Baker’s circuit extractor, developed at MIT. This is a simple keyword format in which every node and every transistor is explicitly listed. Apart from that the format is fairly compact. It is unlikely that anyone will want to create a specification of a circuit in .sim format, except from the circuit extractor.

The other format, a JaM program, is typically used for making incremental changes to a circuit that has been read in from .sim format. Since JaM programs can be executed from files, this format also provides a way of storing such changes, or even whole circuits. An advantage of using the JaM format is that procedural descriptions are available, so that repeated parameterized structures can be efficiently represented.

2.2 Usage

While many different scenarios are possible, one is given here to indicate one way in which SimMOS can be used. SimMOS is first obtained and initialized by executing the relevant command file for the machine being used. A circuit specification, typically obtained from the circuit extractor is read from a .sim file into the internal data structures of SimMOS. Initially, to check out some small part of the circuit, certain named nodes might be set to chosen values (hi or lo) using JaM commands, and the circuit solved. (This can take anywhere between instantaneous for a small circuit, through a few seconds for a large circuit on a Dorado, to a few minutes for a large circuit on an Alto). Other node values can now be examined individually to determine whether the right behaviour was obtained. If anomolous behaviour is observed, the cause can usually be found by running other small tests and looking at other nodes, and then the circuit can be reconfigured interactively.

It soon gets tedious having to set and examine individual node values. Consequently a few utilities are provided to deal with sequences of values for sets of nodes. One of these allows a vector of input node names to be defined, together with a string of values for each node name. A similar vector of output node names can be defined. The procedure *arraysim* takes these vectors and for each set of values in the input vector it sets up those inputs, solves the circuit, and collects the outputs named in the output vector. This mechanism allows sets of test patterns to be applied to a circuit. However, depending on the size of circuit and the length of the test strings each such test can take a considerable amount of time. To relieve some of this burden, it is possible to break such a test from the keyboard, examine the output strings collected thus far, modify the circuit/input vectors/node values/etc., and then to resume the test where it left off. Of course, the test can be restarted from the beginning if desired.

When a session is over, the entire state of SimMOS can be saved, including the state of an interrupted test for resumption at some later date. Much of this state is saved automatically in JaM's virtual memory which is always saved and restored from run to run. However, the circuit is held in the SimMOS data structure and if it has been modified then it must be saved separately. This can be done in a number of ways. First, a new .sim file can be written. Alternatively, a JaM file whose subsequent execution will recreate the circuit, can be written. Both of these methods can consume large amounts of disk space, and are inefficient if only a small part of the circuit has been modified. With foresight it is possible to automatically log all JaM commands that modify the circuit, using the *logchanges* command. Given this log the modified circuit can be reconstructed by reading whatever definition was read on the previous session, and then replaying the log of commands that modified the circuit. This method has the added advantage of allowing backing up to previous versions of the circuit, by removing the last part of the log file using a text editor.

Other scenarios are possible. For example, using the ability to mix switch level and functional simulation it is possible to read in a JaM specification of the functional behaviour of a system. Selected parts of the system can then be disabled, and replaced with circuit specifications that implement the disabled parts. In this way, individual pieces of a system can be diagnosed independently and efficiently.

2.3 Where to find SimMOS

SimMOS can be obtained and initialized by executing one of the command files:

```
[Ivy]<DA>Alto>SimMOS>SimMOS.cm
[Ivy]<DA>DStar>SimMOS>SimMOS.cm
```

for the XM Alto and D machine versions respectively. Also on that directory is a file:

[Ivy]<DA>SimMOS>cleanupSimMOS.cm

that will delete most of the files created during a run. This is useful for cleaning up after using a Dorado.

3. COMMANDS

The available commands are from three sources - commands directed to JaM, primitive SimMOS commands, and JaM procedures. While users need not be concerned with these distinctions, it may be helpful to note that online documentation of most commands that are implemented as JaM procedures can be obtained using the JaM utility command "?", e.g. (arraysim)?. Moreover, these commands can be used as a starting point for new commands tailored to a particular need.

The commands are presented here in six categories: Initialization, Circuit Definition, Setting and Reading Values, Running a Simulation, Saving and Restoring State, and Other.

3.1 Initialization

First get and initialize SimMOS as described under 2.3 "Where to find SimMOS". From here on the operating procedures for both Alto and D machine versions are the same. SimMOS is normally started by typing:

```
jam simmos
```

When loaded SimMOS should prompt with the message:

```
SiMMOS
>>
```

at which time it is ready for commands. The initial getting and initializing if SimMOS will leave SimMOS in this same state.

Note that commands in JaM are executed when a carriage return is typed. Several commands can be put on one line.

simreset
Usage: **simreset**
Initialize the stored circuit to null. The circuit is automatically initialized on start-up, so simreset need be called only to clean out an existing circuit. e.g.

```
simreset
```

Fine point: while simreset is implemented in all versions, it doesn’t currently release the space used in all versions, so its use for large circuits is to be discouraged - get out of SimMOS and start again.

3.2 Circuit Definition

simread
Usage: <filename> **simread**
Read the definition of a circuit in .sim format from the indicated file. e.g.

```
(mycircuit.sim)simread
```

.run
Usage: <filename> **.run**

Run the indicated file as a JaM program. This JaM primitive is included here since it is used for creating a circuit specified as a JaM program. e.g.

```
(mycircuit.jam).run
```

simnode

Usage: <nodename> **simnode**

Create a node having the given name. If a node of the given name already exists this does nothing. Note that the nodes (VDD) and (GND) are always defined in SimMOS, and have the expected values. Node names are stored as originally specified, but their recognition is case shift independent. For example (vdd), (Vdd), and (VDD) are all equivalent. e.g.

```
(newnode)simnode
```

would create a node having the name "newnode".

etrans

Usage: <gate><source><drain> **etrans**

Create an enhancement mode transistor connecting the nodes having the specified names. Any node that doesn’t already exist is created. e.g.

```
(input)(GND)(pulldown)etrans
```

dtrans

Usage: <gate><source><drain> **dtrans**

Create a depletion mode transistor connecting the nodes having the specified names. Any node that doesn’t already exist is created. e.g. to create a pullup for node bus0:

```
(bus0)(bus0)(VDD)dtrans
```

listetrans

Usage: <gate><term1><term1> **listetrans**

List all enhancement mode transistors having the specified gate, and source and drain being term1 and term2 in either order. Output is in the form of a list of etrans commands. A null string for a node name acts as a wild card, e.g. to list all enhancement mode transistors (not to be used indiscriminately):

```
()()()listetrans
```

or to list all enhancement mode transistors with gates connected to the node PHI1:

```
(PHI1)()()listetrans
```

listdtrans

Usage: <gate><term1><term1> **listdtrans**

Same as listetrans except that only depletion mode transistors matching the given names are listed.

Note that depletion mode pullups are not represented explicitly in the SimMOS data structure. Consequently super buffer pullups will be reported as regular depletion mode pullups. Other types of depletion mode transistors, such as in function blocks, are correctly reported.

listtrans

Usage: <gate><term1><term1> **listtrans**

A combination of listetrans and listdtrans.

redefinetrans

Usage: <gate><term1><term1> <gate><source><drain> **redefinetrans**

Redefine the transistor specified by the first three parameters to be connected as given by the last three parameters. The first three parameters are used as in listtrans, using wild cards etc. If the first three parameters do not define a unique transistor then no action is taken. e.g. to reconnect the drain of a given transistor from node (n1) to (n2):

```
(g0)(n0)(n1) (g0)(n0)(n2) redefinetrans
```

logchanges

Usage: <stream> **logchanges**

Arrange to log on <stream> all JaM commands that modify the circuit, for later replaying of such modifications. The stream must be set up using the JaM command .bytestream, e.g.

```
(changesstream)(changes.log) 4 .bytestream .def
changesstream logchanges
```

This opens the file *changes.log* and sets it up for logging changes. The parameter, 4, to .bytestream creates the stream for appending. This is useful for adding to an existing log of changes. Use of 6 instead would rewrite or create the file afresh.

unlogchanges

Usage: **unlogchanges**

Switch off logging of changes.

3.3 Setting and Reading Values

circuitreset

Usage: <value> **circuitreset**

Charge every node that is connected to at least one gate to <value>, for <value> = (0) | (1) | (X). (X) indicates "undefined". e.g.

```
(0)circuitreset
```

would charge all gate nodes to low.

hi

Usage: <nodename> **hi**

Tie the indicated node to VDD. e.g.

```
(input1)hi
```

lo

Usage: <nodename> **lo**

Tie the indicated node to GND. e.g.

```
(input1)lo
```

x

Usage: <nodename> **x**

Remove VDD or GND from the indicated node. The value of the node will not change even if it is not a storage node. This is useful for initialization. e.g.

```
(trinode)hi (trinode)x
```

chhi

Usage: <nodename> **chhi**
Charge the indicated node to high. The node must be connected to the gate of at least one transistor for this to have any effect. e.g.

```
(bit5)chhi
```

chlo

Usage: <nodename> **chlo**
Charge the indicated node to low. The node must be connected to the gate of at least one transistor for this to have any effect. e.g.

```
(bit5)chlo
```

gatestorage

Usage: <switch> **gatestorage**
Set mode for charge storage. If switch=.true then charge can be stored only on the gates of transistors. If switch=.false then charge can be stored on any node. In either case, charge cannot be stored on a node that is either an input or is pulled up.

getnodevalue

Usage: <nodename> **getnodevalue** => <nodevalue>
Get the current value of the indicated node as a string. The <nodevalue> is left on the JaM stack as a string = (1) | (0) | (X) . e.g. *(input1)getnodevalue* might leave the string *(1)* on the stack. This string could be printed with the JaM command "=", e.g.

```
(input1)getnodevalue =
```

might print

```
0
```

getbit

Usage: <nodename> **getbit** => <nodevalue>
Get the current value of the indicated node as an integer. The <nodevalue> is left on the JaM stack as an integer = (-1 | 0 | 0) for (hi | lo | x), respectively (notice that x results in 0). e.g. *(input1)getbit* might leave the integer *-1* on the stack. This integer could be printed with the JaM command "=", i.e.

```
(input1)getbit =
```

would print

```
-1
```

This value is useful for subsequent JaM logical operations, such as .bitand, .bitor, .bitnot, .bitxor, for functional simulation.

putbit

Usage: <nodename><value> **putbit**
Set the indicated node to the indicated value. The node is set to lo | hi for <value> = zero | non-zero, respectively. e.g.

```
(input1) 1 putbit
```

would set input1 to hi.

getword

Usage: <wordspecifier> **getword** => <word>
Construct a word from the current values of the nodes defined by <wordspecifier>, and leave that word on the JaM stack. The <wordspecifier> is a JaM array of node names, typically set up using the JaM utilities "[", "]", and ".def", e.g.

```
(aluoutput) [ (out4)(out3)(out2)(out1)(out0) ] .def
```

Spaces around the "[" and "]" are necessary. Then

```
aluoutput getword
```

will leave an integer on the JaM stack made up of the values of the indicated nodes, right justified in the word, i.e. in the above example, out0 will be represented as the lsb of the result.

putword

Usage: <wordspecifier><value> **putword**
Set the nodes defined in <wordspecifier> to the corresponding bits in the integer <word>. The <wordspecifier> is as defined in getword. e.g.

```
(bus) [ (in2)(in1)(in0) ] .def
```

then

```
bus 6 putword
```

would set: in2 to hi, in1 to hi, in0 to lo.

arraynames

Usage: <nodename array> **arraynames**
List the names of the nodes given in <nodename array>. This command is for use with *arraysim* and for looking at *wordspecifiers* (see *getword*, *putword*). e.g.

```
outarray arraynames
```

might generate

```
(out1)(out2)
```

arraystrings

Usage: <nodename array> **arraystrings**
List the current values of all the node strings in <nodename array>. This command is for use with the command *arraysim*. More than one array can be listed by giving two commands consecutively, e.g.

```
inarray arraystrings  outarray arraystrings
```

might generate

```
01010101 = in0
00110011 = in1
00001111 = in2

01101001 = out1
```

```
    10110101 = out2
```

If each string will not fit on one line then the whole set is continued below. This is to make it easier to establish correspondence between values.

arrayvalues

Usage: <nodename array> **arrayvalues**

List the current values of the nodes given in <nodename array>. This command is for use with *arraysim* and for looking at *wordspecifiers* (see *getword*, *putword*). e.g.

```
    outarray arrayvalues
```

might generate

```
    out1 = 1
    out2 = 0
```

3.4 Running a Simulation

simstep

Usage: **simstep**

Run the simulator to solve the circuit for the current set of input values. See also *maxmicrosteps*. In case of long or unterminating computations, simstep can be interrupted using the JaM break key (rightshift-swat). Fine point: currently it is inadvisable to hold the break key down for an extended period, or you will get a stack overflow. Better to strike the key firmly, then have patience, since the break will not actually happen until the next microstep has completed.

simsolve

Usage: **simsolve** => <number of iterations>

Same as simstep except that the number of iterations needed to achieve convergence is left on the JaM stack. Zero iterations means that the state of the circuit did not change. This is useful for including functional simulation. See also *maxmicrosteps*.

maxmicrosteps

Usage: <number> **maxmicrosteps**

Set maximum number of iterations that will be attempted in simstep and simsolve. The default is 100.

microstep

Usage: **microstep** => <#changes>

Run one iteration of the simulator, and leave the number of nodes that changed on the stack.

reportshorts

Usage: <boolean> **reportshorts**

Control reporting of shorts in microstep. Default is .true.

reportchanges

Usage: <boolean> **reportchanges**

Control reporting of changes in microstep. Default is .false. If .true then nodes that change during a microstep are listed. This is useful for diagnosing circuits that do not stabilize.

arraysim

Usage: <inputarray><outputarray> **arraysim**

For each set of values in the input vector, set up the inputs, solve the circuit, and collect the outputs named in the output vector. The input and output arrays are JaM arrays of node names, typically set up using the JaM utilities "[", "]", and ".def", e.g.

```
(inarray) [ (in0)(in1)(in2) ] .def
(outarray) [ (out1)(out2) ] .def
```

Each of the names specified in the input array must be defined as JaM strings representing the sequence of values required for each input node, e.g.

```
(in0)(01010101).def
(in1)(00110011).def
(in2)(00001111).def
```

The node names specified in the output array will be associated with the collected output strings and need not be initialized. Given the above definitions, the command:

```
inarray outarray arraysim
```

will solve the circuit 8 times, having set up the inputs in2, in1, in0 to 000, 001, 010, 011, 100, 101, 110, 111, and will collect the outputs at each step as strings in the JaM variables out1 and out2. These strings can be examined using the JaM command =, e.g.

```
out1 =
```

might generate

```
01101001
```

It is often convenient to have all the inputs and outputs listed together. See the commands *arraystrings*, *arrayvalues*, and *arraynames* under 3.3.

While various clocking schemes etc. can be implemented using arraysim as described, it is sometimes more convenient to be able to do several things between setting up each set of inputs and collecting the corresponding outputs. For this reason, arraysim actually works as follows. For each set of values in the inputarray arraysim does the following 4 steps:

1) set up the input values,
2) invoke the JaM procedure *clock1*,
3) collect the outputs, and,
4) invoke the JaM procedure *clock2*.

At initialization the procedure clock1 is set up to simply call simstep, and the procedure clock2 is a nop. These procedures can be redefined to implement more elaborate schemes. For example, to implement a 2 phase, non-overlapping clock, PHI1 and PHI2, to be cycled after each input is set up, clock1 could be redefined as follows:

```
(clock1)
((PHI1)hi simstep
 (PHI1)lo simstep
 (PHI2)hi simstep
 (PHI2)lo simstep
).cvx .def
```

Incidentally, for this example it is normally adequate, and more efficient, to define clock1 to be:

```
(clock1)
((PHI1)hi (PHI2)lo simstep
 (PHI1)lo (PHI2)hi simstep
).cvx .def
```

arraysim can be interrupted during operation by pressing the JaM break key (rightshift-swat). Fine point: currently it is inadvisable to hold the break key down for an extended period, or you will get a stack overflow. Better to strike the key firmly, then have patience, since the break will not actually happen until the next simulation cycle has completed. Having interrupted arraysim, almost all of the commands can be safely used for e.g. examining node values, setting node values, modifying the circuit, changing the input array strings (but not the list of names in the input and output arrays), and even for leaving SimMOS completely. The simulation can be restarted using *resumearraysim* (see below).

resumearraysim
Usage: **resumearraysim**
Resume the running of an arraysim after it has been interrupted.

3.5 Saving and Restoring State

simwrite
Usage: <filename> **simwrite**
Write a .sim file describing the circuit currently stored in the SimMOS data structure. e.g.

```
(newcircuit.sim)simwrite
```

A circuit thus saved can be restored later using *simread*.

writestate
Usage: <filename> **writestate**
Write a JaM file describing the current state of the circuit stored in the SimMOS data structure. This file takes the form of a list of invocations of hi,lo,x,chhi,chlo. e.g.

```
(circuitstate.jam)writestate
```

Subsequent use of the .run command on this file will cause the state of all the nodes to be restored. After such a restoration, one call to simstep should be made, to drive non-storage nodes to their correct values.

3.6 Other

This section contains miscellaneous commands from JaM and SimMOS that might be found useful.

inverter
Usage: <inputnode><outputnode> **inverter**
Create a pair of transistors implementing an inverter for the given pair of nodes.

nand
Usage: <in1><in2><out> **nand**
Create 3 transistors to implement a nand circuit.

nor
Usage: <in1><in2><out> **nor**
Create 3 transistors to implement a nor circuit.

pullup

Usage: <nodename> **pullup**

Create a depletion mode pullup transistor to pull up the indicated node. e.g.

```
    (bus0) pullup
```

gensym

Usage: **gensym** => <uniquename>

Leaves a unique node name as a string on the JaM stack. This is useful for generating names of interior nodes in procedurally defined circuits.

gennames

Usage: <string of names> **gennames**

Declares JaM variables having the given names, and assigns to them their own names concatenated with the result of a single common call to gensym. This command is useful in procedural definitions of circuits in which local names that can later be identified are needed. e.g.

```
    (in out int)gennames
```

generates the 3 JaM variables, in, out, and int, with the string values (in#), (out#), and (int#), respectively, where # is the result of a single call to gensym.

/getargs

Usage: <a1><a2>...<an>(<name1> <name2>...<namen>) **/getargs**

This JaM command expects a string of names separated by spaces. It creates JaM variables having the given names, and assigns to them the values preceding the string on the stack in a one to one correspondence. e.g.

```
    3 1.5 (hello)  (i r s) /getargs
```

would create the JaM variables i,r, and s, and assign to them the values 3, 1.5, and the string hello, respectively. This command is useful as the first command in a JaM procedure, to get the arguments from the stack.

.print

Usage: <string> **.print**

This JaM command prints the given <string>. e.g.

```
    (hello).print
```

/print

Usage: <string> **/print**

This JaM command prints the given <string> followed by a carriage return.

.def

Usage: <name><value> **.def**

This is the JaM assignment command, and assigns <value> to <name>. .def is used to set up both ordinary variables and procedures. e.g.

```
    (numbervariablename) 23 .def
    (stringvariablename)(this is a string, not a procedure) .def
    (procedurename)((hello).print).cvx .def
```

.cvx

Usage: <object> **.cvx**

This JaM command changes the execution interpretation of the given <object>. When applied to a string, subsequent execution of that string will cause it to be interpreted.

.rept

Usage: <number><object> **.rept**
This is the JaM iteration command, and will execute the given <object> <number> times.
e.g.

```
5 (procedurename (!).print).cvx .rept
```

would print, (given the definition of *procedurename* given under the description of .def):

```
hello!hello!hello!hello!hello!
```

.quit

Usage: **.quit**
This is the JaM termination command, and must be used for leaving JaM in a controlled way with virtual memory saved.

4.0 Examples

First get and initialize SimMOS as described under section 3.1 Initialization.

4.1 Nor Gate

Simulation of this simple circuit will be covered in some detail to show the various possibilities. The circuit is a simple 2 input nor gate. The inputs will be nodes *in1* and *in2* and the output will be node *out*. First of all it is necessary to define the circuit. In .sim format this might be represented on the file *norcircuit.sim* by:

```
e in1 gnd out
e in2 gnd out
d out out vdd
```

and in JaM:

```
(in1)(GND)(out)etrans
(in2)(GND)(out)etrans
(out)(out)(VDD)dtrans
```

This latter definition could be typed directly to SimMOS or could be put on a file, say *norcircuit.jam*.

We now start SimMOS. This is done by typing

```
jam simmos
```

to the Alto executive, and waiting for the prompt ">". If we wish to read the circuit definition from the .sim file, type:

```
(norcircuit.sim)simread
```

or to read it from the JaM file:

```
(norcircuit.jam).run
```

or you may just want to type it in directly as JaM commands. You may want to convince youself of the presence of the circuit. Try:

```
()(GND)()listtrans
```

to get a list of all transistors with either source or drain connected to GND. Let us now carry out a simple test of the circuit. Set in1 high and in2 low:

```
(in1)hi (in2)lo
```

Now solve the circuit:

```
simstep
```

and look at the output:

```
(out)getnodevalue =
```

which should print

```
0
```

The setting of individual inputs like this rapidly gets tedious, so let us create a set of test vectors. First define an array, *inarray*, of names of nodes to be considered inputs:

```
(inarray) [ (in1)(in2) ] .def
```

Spaces around "*[*" and "*]*" are needed. Now define the JaM variables *in1* and *in2* to be the required sequences of values:

```
(in1)(0011).def
(in2)(0101).def
```

Note that the names in1 and in2 are serving two purposes: as node names in SimMOS, and as string variables in JaM. However, no confusion should arise. Now set up an array of outputs to be collected. In this case it has only one entry:

```
(outarray) [ (out) ] .def
```

Now a sequence of simulations can be run:

```
inarray outarray arraysim
```

We can now look at the outputs using *arraystrings*. It is convenient to have the inputs and outputs listed together, so type two commands together:

```
inarray arraystrings  outarray arraystrings
```

which should produce:

```
0011 = in1
0101 = in2

1000 = out
```

We may now want to save the state of the circuit and terminate the session. To save the values of the various nodes do:

```
(norcircuit.state)writestate
```

Later on during this session, or in a subsequent session, the state of the circuit can be restored by:

```
(norcircuit.state).run
```

To leave SimMOS in a controlled way, type:

```
.quit
```

4.2 Shift Register

We shall define a 4 stage shift register in JaM. It is built up of 4 instances of a shiftcell, which is itself built up of 2 instances of a halfshiftcell:

```
(halfshiftcell)
((hsc.clock hsc.in hsc.out)/getargs
 (hsc.internal) gennames
 hsc.clock hsc.in hsc.internal etrans
 hsc.internal hsc.out inverter
).cvx .def

(shiftcell)
((sc.in sc.out)/getargs
 (sc.internal) gennames
 (phi1) sc.in sc.internal halfshiftcell
 (phi2) sc.internal sc.out halfshiftcell
).cvx .def

(in)(s0)shiftcell
(s0)(s1)shiftcell
(s1)(s2)shiftcell
(s2)(s3)shiftcell
```

The /getargs command assigns the relevant number of arguments to the JaM variables given. Now set up input and output node arrays:

```
(inarray) [ (in) ] .def
(outarray) [ (s0)(s1)(s2)(s3) ] .def
```

and define a test vector for *in*

```
(in)(010011000111).def
```

Note that we could have defined the two phase clocks, *PHI1* and *PHI2*, as inputs. However, since we are interested in what results after each whole clock cycle we shall do the clocking in the *clock1* procedure. Therefore redefine *clock1* to be:

```
(clock1)
((phi1)hi (phi2)lo simstep
 (phi1)lo (phi2)hi simstep
).cvx .def
```

Now run a simulation and look at the results:

```
inarray outarray arraysim
inarray arraystrings  outarray arraystrings
```

Incidentally, since the above three commands are often used it may be worth setting up a JaM command to do them. This should produce:

```
010011000111 = in

010011000111 = s0
X01001100011 = s1
XX0100110001 = s2
XXX010011000 = s3
```

Notice the unknowns, X, that in this case get flushed out of the circuit. Unknowns can be a source of great difficulty in SimMOS, especially in circuits involving feedback, where such unknowns may never get flushed out. The more recent versions of SimMOS are able to correctly resolve a large class of problems involving propogation of unknowns. However the technique works on a single microstep basis, so that unknowns that can be resolved only by analyzing several microsteps will not be eliminated.

Several strategies have been developed for dealing with unknowns. First, if your circuit has a *reset* line then use it. Indeed, it may well be worth considering a reset line on any circuit anyway, as part of your design philosophy, just to be sure that the circuit can always be set to a known state. However, this is not always desirable. In such cases, another remedy that has been used is to build a definition of a reset circuit to be used together with the circuit under test. This can consist of transistors that gate VDD or GND onto selected nodes. Another method that has been used successfully by Jim Cherry applies to two-phase clocked circuits. Simply set both phases of the clock high and run a *simstep*. This "opening of the sluices" usually flushes out any unwanted X's. Perhaps the most obvious remedy is to build a JaM procedure that uses *chhi* and *chlo* to charge or discharge specific nodes. Finally, the procedure *circuitreset* can be used to set all charge storing nodes to some given value. Since this normally leaves the circuit in an inconsistent state it is usually best to follow it with a simstep.

4.3 Two Phase Clock Generator

While SimMOS is basically designed to handle combinatorial circuits, possibly including charge storage, it turns out that the iterative solution used has the effect of modelling every transistor as a switch having a unit delay. Consequently some timing dependent circuits can be simulated. Indeed, the detailed behaviour of a circuit modelled as unit delay switches can be observed by using the *microstep* command. (This command is used by *simstep* to achieve a converged solution).

As an illustration of the simulation of such circuits, consider the two phase non-overlapping clock generator given in [Mead and Conway] in Figure 7.6(b) on page 229. While this may not be the best way to generate clocks for SimMOS, it is important that SimMOS can handle it in case one is using such a generator as part of an existing circuit. The circuit can be modelled as:

```
(int) gensym .def
(clk) int inverter
(clk)(phi2)(phi1) nor
(phi1) int (phi2) nor
```

where *int* is the one internal node, *clk* is the input single phase clock, and *phi1* and *phi2* are the generated output clocks. You can readily verify that this circuit behaves properly at the *simstep* level, by trying the two possible values of *clk*, running a *simstep* and looking at the generated clocks. However, it may be interesting to examine the behavior at the *microstep* level, as follows. Define the *clock1* procedure to be:

```
(clock1)
(microstep .pop
).cvx .def
```

and set up the input and output arrays:

```
(inarray) [ (clk) ] .def
(outarray) [ (phi1)(phi2) ] .def
(clk)(000001111100000111110000011111).def
```

This is a case where it is necessary to initialize the node values, so do:

```
(0)circuitreset
simstep
```

After running an *arraysim* the result should be:

```
000001111100000111110000011111 = clk

111110000000011100000000111000000 = phi1
000000111110000011111000001111 = phi2
```

This exercise can be extended to make a ring oscillator to generate the single phase clock *phi1*. Set up 4 inverters:

```
(clk)(a) inverter
(a)(b) inverter
(b)(c) inverter
(c)(d) inverter
```

and initialize them:

```
simstep
```

Now connect the last one to the first with another inverter, thereby making a 5 stage ring oscillator, and stop driving *clk* externally (left over from previous simulation):

```
(d)(clk) inverter
(clk) x
```

Take care not to issue a *simstep* command now, since the circuit will never converge. This clock generator can now be run for, say, 10 cycles, printing the values of *clk*, *phi1* and *phi2*, as follows:

```
10
(microstep .pop
 (clk)getnodevalue .print ( ).print
 (phi1)getnodevalue .print
 (phi2)getnodevalue /print
).cvx .rept
```

which should generate

```
0 01
0 01
0 00
0 10
0 10
1 10
1 00
1 01
1 01
1 01
```

This last example is included only to illustrate something of the operation of SimMOS. To simulate a circuit of any size by such a method would be inefficient, and would probably give misleading results since time is not modelled at all realistically in SimMOS.

4.4 Functional Simulation

<"in preparation">

5.0 References

[JaM] JaM, John Warnock and Martin Newell, [Ivy]<JaM>JaM.bravo.

[Mead and Conway] *Introduction to VLSI Systems*, Carver Mead and Lynn Conway, Addison Wesley, 1980.

6.0 Index of Commands