379
N81
NO. 6539

IMPROVING DIGITAL CIRCUIT SIMULATION:
A KNOWLEDGE-BASED APPROACH

THESIS

Presented to the Graduate Council of the
University of North Texas in Partial
Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

John A. Benavides, B.S.E.E.

Denton, Texas

August, 1989

Benavides, John A., Improving Digital Circuit Simulation: A Knowledge-based Approach. Master of Science (Computer Science), August, 1989, 78 pp., 34 illustrations, bibliography, 37 titles.

This project focuses on a prototype system architecture which integrates features of an event-driven gate-level simulator and features of the multiple expert system architecture, HEARSAY-II. Combining artificial intelligence and simulation techniques, a knowledge-based simulator was designed and constructed to model non-standard circuit behavior. This non-standard circuit behavior is amplified by advances in integrated circuit technology. Currently available digital circuit simulators can not simulate this behavior. Circuit designer expertise on behavioral phenomena is used in the expert system to guide the base simulator by manipulating its events to achieve the desired behavior.

Copyright by
John Anthony Benavides
1989

ACKNOWLEDGMENTS

The author wishes to express his deepest appreciation and thanks to members of his thesis committee, Dr. Mahir Ali, Dr. Dana Wyatt, and Dr. Kathleen Swigger, for their patience, encouragement, and financial support over the years. Financial support from the Special Interest Group on Design Automation (SIGDA) of the Association for Computing Machinery is gratefully acknowledged. Finally, special thanks to my family, Josefina and Jesse, for their love and support.

TABLE OF CONTENTS

	Page
LIST OF ILLUSTRATIONS	vii
Chapter	
I. INTRODUCTION	1
Knowledge-base Simulators	
SWIM Prototype	
II. SWIM SYSTEM ARCHITECTURE	6
System Components	
World Data Base (WDB)	
Types of WDB Information	
Sources of WDB Information: Engineer's Design Process	
Storage of Information: Data Structures	
WDB Example Circuit	
Model Base	
Simulator	
Event Queue and Event Queue Manager	
Time Manager	
Clock	
Evaluator	
Expert System	
Knowledge Sources	
Blackboard	
SWIM's Dynamic Behavior	
SWIM User Interface	
Summary	

Chapter	Page
III. SYSTEM EXAMPLE	36
The Circuit	
First Subcircuit	
Second Subcircuit	
Knowledge Sources	
Switching-pads KS	
Multiplexer KS	
DRAM Refresh KS	
Traces	
Initialization Trace	
Normal Evaluation Trace	
DRAM Refresh KS Trace	
Switching-pads KS Trace	
Multiplexer KS Trace	
Summary	
IV. CONCLUSION	69
APPENDIX	73
REFERENCES	75
SELECTED BIBLIOGRAPHY	78

LIST OF ILLUSTRATIONS

Figure		Page
1.	SWIM system architecture	7
2.	Parts, pins, and nets	8
3.	LISP structures for world data base	13
4.	Circuit for 2 to 1 Multiplexor	15
5.	Dump of WDB for circuit in Figure 4	17
6.	Model of an AND gate	18
7.	Expanded SWIM architecture	20
8.	Example of a TREE domain definition for circuit of Figure 4 . .	25
9.	Example of a SET type domain definition	25
10.	Multiple TREE type domain definition for circuit of Figure 4 . .	26
11.	Knowledge source definition record	28
12.	Knowledge source instantiation information	30
13.	Evaluator pseudo code	31
14.	SWIM simulator window	33
15.	Example Circuit	37
16.	Input signal whose value is used on multiplexer outputs	40
17.	ROWSEL truth table	41
18.	RAMCELL operation table	42
19.	Domain definition for switching-pads KS	44
20.	Ignition statement for switching-pads KS	44
21.	Rule base for switching-pads KS	46
22.	Behavior of AND, OR, and NOT gates	47

23.	Simple circuit with reconvergent fanout	48
24.	Domain definition for multiplexer KS	50
25.	Simulator and KS evaluated behavior for multiplexer circuit . .	51
26.	Domain definition for multiplexer KS	51
27.	Rule base for multiplexer KS	53
28.	DRAM refresh KS	55
29.	Initialization trace	58
30.	Normal evaluation trace	61
31.	DRAM refresh KS trace	62
32.	Switching-pads KS trace	65
33.	Multiplexer KS trace	67
34.	LISP model for RAMCELL	74

CHAPTER I

INTRODUCTION

Simulation in the electrical engineering domain serves multiple purposes. With a simulator, an engineer can analyze a circuit before investing time and money in manufacturing. Once correct functionality of the design is established, an engineer can also use the simulator to check the circuit's performance speeds, as well as the performance of manufacturing tests (Terman 1987, Trimberger 1987).

Simulators currently available for digital circuit design work at varying levels of circuit abstractions. The lowest level of abstraction would model the circuit using the physics of its components (Nagel 1975). While this type of continuous simulation is very accurate, the computation costs are high, even for small circuits containing no more than several thousand circuit devices (Terman 1987). Simulators with higher levels of abstraction model the circuit at the switch level, logic gate level, functional block level, or at a behavioral level using a discrete simulation methodology (Dillinger 1988, Lightner 1987). As the level of abstraction increases, computation costs decrease but so does simulation accuracy (Miczo 1986).

Another complication occurs as the speed of circuit components increases because circuit behavior breaks from clean digital switching characteristics. The problems of simultaneously switching outputs, transmission line reflections, power supply noise, and crosstalk between signals become major design concerns (Texas Instruments 1987). To simulate these problems requires that an engineer be an

expert with both the phenomena and modeling it or use simulation tools which recognize such phenomena.

Several methods of modeling these phenomena are possible. The first would require a reduction of the entire circuit to a physics level model abstraction. A continuous simulator would then exhibit the non-standard digital circuit phenomena. However, because of the prohibitive computation costs, this can not be done with large circuits. A second method would be to combine the abstraction levels, a so-called mixed-mode simulator (Bloom 1987). By simulating portions of a circuit at the physics level and portions at a digital circuit level, a designer might be able to observe these phenomena. However, this method requires that the designer be an expert with digital circuit phenomena in order to recognize where to partition the circuit. In addition, the mixed-mode technology is still young and unproven in its capability (Milne 1989). The research reported in this paper is aimed at pursuing a third method using new simulator technology that combines artificial intelligence techniques with simulation. It does not require expert knowledge of circuit phenomena. This hybrid method encapsulates expert knowledge of known phenomena in a knowledge-based simulator in order to model non-standard behavior and avoid the difficulties of the previous methods.

In this paper, a prototype knowledge-based simulator will be reported which combines an event-driven digital circuit simulator with a knowledge-based expert system. The event-driven digital circuit simulator brings with it efficient analysis of coupled components behavior. This is standard behavior and results from formal connections. The expert system recognizes, analyzes, and initiates the simulation of behavior resulting from informal connections. This is possible because such non-standard behavior may be linked to physical, structural, or topographical information not ordinarily used by the digital circuit simulator. The expert system steers

the simulator into non-standard behavior by modifying its events during operation, resulting in future behavior which reflects a circuit's reaction to the initial non-standard behavior.

The knowledge-based simulator discussed herein is envisioned as a design tool which could be used early in the design process. By discovering and understanding non-standard behavior, a designer could incorporate appropriate circuit design practices in order to avoid potential problems and prevent the expense of manufacturing a non-functional circuit. The following section explores previous research aimed at combining knowledge-bases and simulation.

Knowledge-based Simulators

As artificial intelligence (AI) has matured as a discipline, applications of AI have migrated into other disciplines of computer science. Simulation is one field which has garnered new techniques from this growth (O'Keefe 1986). Murray and Sheppard (1988) note that knowledge-based simulation systems employing artificial intelligence techniques have been designed for several purposes. These include:

- 1) Computer-assisted model construction.
- 2) Decision-support functions.
- 3) Intelligent front-ends for assisting in simulation experimentation and results analysis.
- 4) Complete simulation environments for supporting the simulation model life cycle.

Two approaches have been used to achieve these capabilities.

One approach has been to use a knowledge-based expert system as the base system. Murray and Sheppard (1988) built a computer-assisted model constructor for SIMAN models by using an expert system containing knowledge about model construction techniques and SIMAN. Moser (1986) implemented a decision-support system using an expert system knowledgeable on corporate business to analyze simulation results of a corporate model. Borden et. al. (1985) incorporated temporal knowledge into an OPS5 expert system in order to simulate electronic warfare threats. In general, system architectures which use expert systems as their base can represent knowledge in rule bases and through inference strategies.

The second design approach for knowledge-based systems has been to use an object-oriented programming paradigm. Kim and Zeigler (1988) note that this approach is popular because of its similarity to the discrete event formalisms. In object-oriented programming, objects represent real world entities which contain methods and properties. Methods are behavior rules used to respond to messages representing world changes. All communication between objects is accomplished with messages which may represent passage of time, changes in object relations, or requests for object action. If an object does not contain a method for a specific message, it may inherit a method from an object defined to be superior to it in its object hierarchy (Adelsberger et al. 1986). One example of this approach is ROSS, which was designed by the RAND corporation (Klahr and Waterman 1986). ROSS supports the simulation model life cycle and has been applied to military battle simulations (Klahr, McArthur, and Narain 1982, Klahr 1986). KBS is a second example of an object-oriented system (Reddy et al. 1986). KBS is a decision support environment for intelligent management of business operations. Simulation, model verification, and result analysis are all parts of the KBS environment. In general, systems using object-oriented programming represent knowledge through

methods, object properties, and inheritance.

SWIM Prototype

A prototype system called SWIM, Simulate What I Mean, has been constructed on a Texas Instrument's Explorer Workstation in LISP at the University of North Texas. SWIM integrates features of an event-driven gate-level simulator (Lightner 1987, Dillinger 1988) and features of the multiple expert system architecture, HEARSAY-II (Hayes-Roth, Waterman, and Lenat 1983). Using this hybrid architecture, simulation of a digital circuit is controlled through the Explorer's interactive windowing system.

The approach used in the SWIM prototype differs from the expert system shell and object-oriented language approaches because it integrates the expert system directly into the simulator. The expert system becomes a controlling component of the base simulator in order to increase simulation capability.

The remainder of this paper discusses the design, implementation, and analysis of the SWIM system. Chapter 2 presents the system architecture. Chapter 3 illustrates the use of the SWIM prototype. The paper concludes with implementation notes and comments on future enhancements.

CHAPTER II

SWIM SYSTEM ARCHITECTURE

This chapter describes SWIM's system architecture. It begins by describing the functions of the major components of the system, followed by a description of the system's dynamic behavior. Finally, the chapter ends with a section describing SWIM's user interface.

System Components

The five major components of the system architecture are the world data base, the model base, the simulator, the expert system, and the evaluator (Figure 1). The world data base contains all the information on the circuit being simulated. Both the evaluator and the expert system interrogate the world data base. The model base contains simulation models. The simulator coordinates events, manipulates system time, tracks system operation, and schedules behavioral evaluation with the evaluator. The evaluator determines behavior through model analysis and expert system consultation. When expert knowledge is available from the expert system, the evaluator modifies its behavioral interpretation based on model analysis.

In the following sections, the world data base, model base, simulator, expert system, and evaluator will be described in detail.

Figure 1. SWIM system architecture

World Data Base (WDB)

The central component of SWIM is the world data base (WDB); all simulation processes depend on its contents. In this section the WDB's information will be described from three views: content, source, and storage. Next, a simple circuit and its corresponding WDB representation will be described.

Types of WDB Information

When a circuit is simulated, its corresponding knowledge is added into the WDB. The WDB describes the circuit being designed to the simulator and the expert system. There are four types of information in this WDB: structural,

functional, physical, and dynamic. Both the simulator and the expert system can access all four types of information. The simulator uses the WDB to evaluate a circuit's standard performance. The expert system uses the WDB as a source of facts to justify inferences suggesting circuit behavior underivable by the simulator.

Structural information describes the connections (coupling) between circuit. Electrical engineers build systems from primitive or composite parts. These parts are connected in graph-like structures known as circuits. A *part* performs some operation depending on the values present on its *pins*. For example in Figure 2, the part AN1 has three pins. These pins can be inputs, outputs, or a combination of both. A *net* is a collection of pins which are connected together. For example in Figure 2, pin 3 of AN1 and pin 1 of OR1 are connected on net NET2. Pins define the data flow relationship between a net and a part.

Figure 2. Parts, pins, and nets

Functional information describes a part's behavior during simulation. This information is stored in a simulation model. A simulation model describes the behavior of a part's output pins when its input pins are stimulated. In SWIM,

composite parts are logic gates. Behavior models for these parts are described with LISP¹ functions. In Chapter 3, several models and the protocol for constructing models will be described.

In the simulation field, simulationists consider the entire system being simulated as a model. While in the electrical engineering field, the system being simulated is constructed from composite parts which are modeled. Thus, the behavior of the entire electrical system is derived from the combined behavior of the composite parts.

Physical information describes the planned geometric properties of the final circuit. This information is used for manufacturing the system. The system may be an integrated circuit (IC) or an electronic board. For an IC, the physical information describes the many levels of polysilicon, silicon, metal, and other materials that define the parts and their interconnections. For an electronic board, the physical information describes the locations of integrated circuits and the routing of copper connecting the integrated circuits' pins. In both cases, routing between pins is done on multiple layers. Using multiple layers simplifies the routing task and the software required to do the routing.

Some physical properties of a system alter the behavior of the system. The delays through the connections between pins is a function of the metal or wire length. For an IC, the delays of individual parts are increased as the size of the entire IC increases owing to increased capacitance. For electrical boards and integrated circuits the number and proximity of parts can have a marked affect on thermal, noise, and power dissipation characteristics of the system. These characteristics can change the anticipated delays through the entire system and create

¹ The dialect of LISP used is Common LISP.

undesired behavior (Dillinger 1988).

Dynamic information describes the states of all nets connecting parts in the circuit as well as any memory devices that may exist in a part. During a simulation, nets and memory devices change their values to one of four discrete values: high, low, unknown, and high impedance. These changes are defined by the functional models of the circuit's parts. When a change occurs on a net, parts on the net are evaluated to determine if further changes (events) will result. If a change occurs in a memory device, the part containing the memory device is evaluated to determine if further changes will result. Any further changes will be stored until their time to occur arrives.

In addition to storing the state of a net, the time of the net's last change is also stored in the WDB. This time field can be used for detecting hazards by the models. In order for a real world part to perform in its ideal behavior, certain constraints must be met. If these constraints are violated, a hazard is said to exist. Usually, simulator models only perform similar to a part's ideal behavior (McCluskey 1986). However, a simulation model should detect when a hazard exists and communicate this information to a circuit designer. Hazard communication is usually done through the simulator interface by printing out warning messages. One type of hazard is the minimum pulse width hazard. In order for a part to respond to a net change, the net must maintain its value for a minimum amount of time. If the net changes too rapidly, the part experiences a minimum pulse width hazard. Hazards can also occur due to changes on multiple part pins. For example, the value on one pin might be required to maintain stability with respect to another pin's value. If it does not, a hazard may result. Two types of this multi-pin hazard are the *setup* time and *hold* time hazards (McCluskey 1986). By saving the net's last change time, hazards can be detected by the simulation model.

Sources of WDB Information: Engineer's Design Process

This section describes the design steps typically taken by electrical engineers in circuit design. Digital electrical engineering design can be crudely separated into three steps: schematic capture, simulation, and physical design. First, schematic capture is used to describe a circuit. Then, simulation is used to verify the design. Ultimately, physical design is used to prepare the circuit for manufacturing. An engineer may need to repeat one or more of these steps to refine the performance of a circuit before completing a design.

An engineer's first step is to draw the circuit using a schematic capture program. This process is similar to computer-aided drafting² programs. It differs from computer-aided drafting in that the engineer uses predefined symbols for parts instead of drawing each part individually. By drawing a circuit, the engineer produces two items. The first item is a data base describing the circuit. The second item is a graphical description typically used to create paper plots of the circuit.

When an engineer has completed describing the circuit using schematic capture, the circuit's behavior must be simulated. Typical simulators use a schematic capture system's data bases to derive a structural description of a circuit. If the circuit does not behave as desired, the engineer will need to return to schematic capture to modify the circuit's structure. Cycling between schematic capture and simulation is a common practice and electrical engineering CAD systems are designed to permit this behavior. When the engineer's circuit simulates the desired behavior, the circuit is ready to be translated to a physical representation.

Systems that create physical representations place the circuit's parts and route wires between the part's pins. Although the tasks of placement and routing are

² Computer-aided drafting is not the same as computer-aided design (CAD).

closely interrelated, they are usually approached as separate tasks because of their complexity (Dillinger 1988). The program performing the placement task may try to optimize a variety of design objectives such as performance, wireability, chip (or board) size, or accommodating simultaneously switching off-chip drivers. The most frequently used placement strategy seeks to minimize overall wireability by positioning tightly interconnected parts close together. The routing task consists of finding paths for wires between the connected pins. Nets connecting the most parts are usually routed first because they require the most routing space. The result of the physical design step is a geometric physical representation that can be used to manufacture the circuit.

Because engineering design is an iterative process, circuit information will change multiple times before the final design is attained. As new information becomes available, simulation is repeated to verify that new design decisions continue to support circuit requirements. For example, before translating a circuit into a manufacturing format, an engineer will usually back annotate physical design information such as wire lengths (in the form of delays) to the simulator. With the wire lengths available, a more accurate simulation will determine if net delays will adversely change the desired circuit behavior.

Storage of Information: Data Structures

SWIM is designed to hold as much engineering design information as possible to provide a rich fact source for justifying expert system inferences. In the WDB the four types of information described in the previous section are stored throughout *pin*, *part*, and *net* records³. Figure 3 illustrates the LISP structures for

³ There is not a one-to-one mapping between the four types of information and the three types of data structure records.

these records. Recall that pins represent the port through which a part is connected to a net. All pins connected to a net are referenced in the pin-list of a net record. Similarly, all pins connected to a part are referenced in the pin-list of a part record. The remainder of this section describes the data held in these LISP structures.

```
;; Printed fields are marked with P.
;; Default values are shown for each field.
(defstruct net
  "Signal net"
  (name 'unknown)      ;P;
  (type 'core)         ;P; CORE or
                      ; peripheral net (IN, OUT, INOUT)
  (value 'x)           ;P; Current net value
  (change-time 0)      ;Time of last change (Absolute time)
  (pin-list nil)       ;All pins connected to this signal net
  (physical-list nil)  ;Physical information
)
;;
(defstruct part
  "Part instance"
  (name 'unknown)      ;P;
  (type 'unknown)      ;P; Points to functional model
  (eval-bit nil)       ;A flag showing pending evaluation
  (pin-list nil)       ;All its own pins; ordered
  (physical-list nil)  ;Physical information
)
;;
(defstruct pin
  "Pin instance"
  (name 'unknown)      ;P;
  (number 0)           ;P; Pin number
  (type 'inout)        ;P; Legal types are: in, out, or inout
  (part nil)           ;P; Points to owning part
  (net nil)            ;P; Points to owning signal net
  (physical-list nil)  ;Physical information
)
```

Figure 3. LISP structures for world data base.

The pin structure contains six fields. The first field contains the pin's name. The second field contains the pin's number, which is unique for each pin on the part to which it belongs. The third field, the pin type, defines information flow through the pin. A pin can have one of three directions from the net to the part: in, out, or inout. The next two fields are links to the pin's part and net. The final field in the pin structure is the physical list that describes the pin's coordinates in the physical design space.

The part structure contains five fields. The first field contains the instance name for this part. The second field defines the part type and is used to determine the part's behavioral model at simulation time. The third field is used for book-keeping during the evaluator's operation. This field is marked to prevent multiple evaluations of the same part due to multiple pin changes on the part. A more thorough explanation of this field is offered in the evaluator section of this paper. The fourth field is a list of pin structures that belong to this part. This pin-list is ordered according to the pin's number field. The last field is the physical list that describes the part's geometric properties in the physical design space.

The net structure contains six fields. The first field contains the name of the net. The second field, net type, specifies whether the net is a core or a peripheral net. In the real circuit, core nets can not be directly stimulated or observed from the exterior of the circuit as can peripheral nets. The next two fields contain dynamic information about the net: the net value and the time it reached the current value. The fifth field is a list containing all the pins connected to the net. The last field is a list describing the geometric coordinates the net follows in the physical design space.

Some additional observations should be noted about the WDB data structures. Each instance of a pin structure will exist in at most two lists: the pin-list

for a net and the pin-list for a part. This follows because a pin can not belong to two different parts or to two different nets. In Figure 3, the values shown in the fields of the structures are default values obtained when an instance of each structure is created. Finally, fields marked with a **P** in Figure 3 are printed when examined by system software as will be shown in Figure 5.

Figure 4. Circuit for 2 to 1 Multiplexor

WDB Example Circuit

Figure 4 illustrates a simple circuit which acts as a 2 to 1 multiplexor. Figure 5 shows a dump of the data structures for the circuit in Figure 4. This dump was created using the LISP interpreter on the TI Explorer. The ">" symbol in the figure is the interpreter's prompt for a command. The variables shown in Figure 5a, part-list and net-list, contain lists of all the *part* and *net* structures (see Figure 3) for this circuit. The LISP accessor functions *first*, *second*, *third*, ..., *seventh* are

used in the next parts of Figure 5 to retrieve items in the part-list and net-list. In Figure 5b, the function *part-pin-list* is used to view the contents of each part's pin list. Thus, the first function call in Figure 5b, (*part-pin-list (first part-list)*), will access the pin-list of the first item in the part-list, *Part{AN1,AND}*, and return a list of its pins. Pin-lists for other parts in the part-list also shown in Figure 5b. Similarly in Figure 5c, the function *net-pin-list* is used to view the contents of each net's pin-list.

During the simulation, records in the WDB are accessed in a net, pin, part, pin, net order. When a net changes value, the parts connected to the net through an input pin are examined for behavioral changes. A model defining the part's behavior is referenced in the part record. Using this model, values for this part's output pins are determined. Consequently, these output pins may change other nets, and the access loop (net, pin, part, pin, net) is repeated.

Model Base

The second major component of the SWIM system is the model base. The model base is a data base containing all part behavioral models. The SWIM prototype has been constructed with gate-level behavioral models. Models representing logic functions AND, OR, NOT, and NAND, as well as memory elements D, SR, and JK flip-flops, are currently available. Increasing the level of part model complexity is planned as a future enhancement. Each part model is a LISP function that is evaluated at simulation time.


```

> part-list
(Part{AN1,AND}      Part{AN2,AND}
 Part{OR1,OR}       Part{N1,NOT})
> net-list
(Net{A,IN,X}   Net{B,IN,X}   Net{E,IN,X}
 Net{NET1,CORE,X} Net{NET2,CORE,X}
 Net{NET3,CORE,X} Net{Y,OUT,X})
-----
> (part-pin-list (first part-list))
(Pin{ONE,1,IN, Part{AN1,AND}, Net{A,IN,X}}
 Pin{TWO,2,IN, Part{AN1,AND}, Net{NET1,CORE,X}}
 Pin{THREE,3,OUT, Part{AN1,AND}, Net{NET2,CORE,X}})
> (part-pin-list (second part-list))
(Pin{ONE,1,IN, Part{AN2,AND}, Net{E,IN,X}}
 Pin{TWO,2,IN, Part{AN2,AND}, Net{B,IN,X}}
 Pin{THREE,3,OUT, Part{AN2,AND}, Net{NET3,CORE,X}})
> (part-pin-list (third part-list))
(Pin{ONE,1,IN, Part{OR1,OR}, Net{NET2,CORE,X}}
 Pin{TWO,2,IN, Part{OR1,OR}, Net{NET3,CORE,X}}
 Pin{THREE,3,OUT, Part{OR1,OR}, Net{Y,OUT,X}})
> (part-pin-list (fourth part-list))
(Pin{ONE,1,IN, Part{N1,NOT}, Net{E,IN,X}}
 Pin{TWO,2,OUT, Part{N1,NOT}, Net{NET1,CORE,X}})
-----
> (net-pin-list (first net-list))
(Pin{ONE,1,IN, Part{AN1,AND}, Net{A,IN,X}})
> (net-pin-list (second net-list))
(Pin{TWO,2,IN, Part{AN2,AND}, Net{B,IN,X}})
> (net-pin-list (third net-list))
(Pin{ONE,1,IN, Part{N1,NOT}, Net{E,IN,X}}
 Pin{ONE,1,IN, Part{AN2,AND}, Net{E,IN,X}})
> (net-pin-list (fourth net-list))
(Pin{TWO,2,OUT, Part{N1,NOT}, Net{NET1,CORE,X}}
 Pin{TWO,2,IN, Part{AN1,AND}, Net{NET1,CORE,X}})
> (net-pin-list (fifth net-list))
(Pin{THREE,3,OUT, Part{AN1,AND}, Net{NET2,CORE,X}}
 Pin{ONE,1,IN, Part{OR1,OR}, Net{NET2,CORE,X}})
> (net-pin-list (sixth net-list))
(Pin{THREE,3,OUT, Part{AN2,AND}, Net{NET3,CORE,X}}
 Pin{TWO,2,IN, Part{OR1,OR}, Net{NET3,CORE,X}})
> (net-pin-list (seventh net-list))
(Pin{THREE,3,OUT, Part{OR1,OR}, Net{Y,OUT,X}})

```

Figure 5. Dump of WDB for circuit in Figure 4. (a) *Top*, part-list and net-list; (b) *Center*, pins on parts; (c) *Bottom*, pins on nets.

Within the simulator framework, the LISP functions that define model behavior follow a specific protocol. These functions require two inputs. The first input is the pin-list for the instance of the part the function is modeling. The second input is a list containing the events which are stimulating the part's pins. From the pin-list, the function can access the net structures connected to all the pins of the part. Using pin values and the stimulating events, the function determines future events and checks for transition hazards. Future events determined by the simulation model are returned in a list to the evaluator. Any hazards found are reported as errors.

```
(defun and_behavior (pins-list)
  "Behavior for a two input/one output AND gate.
  ;; Input pins: 1 and 2
  ;; Output pin: 3
  (let ((gate-delay 5)      ;Delay for signal transition
        (result-trans nil)
        (pin1-value (net-value (pin-net (first pins-list))))
        (pin2-value (net-value (pin-net (second pins-list))))
        (output-net (pin-net (third pins-list))))
    (case pin1-value
      (1 (case pin2-value
            (1 (setq result-trans '1))
            (0 (setq result-trans '0))
            (x (setq result-trans 'x)))))
      (0 (setq result-trans '0))
      (x (case pin2-value
            (1 (setq result-trans 'x))
            (0 (setq result-trans '0))
            (x (setq result-trans 'x)))))
    ;; Return a list containing the resulting transition
    (list (make-event :time gate-delay
                      :transition result-trans
                      :net output-net))))
```

Figure 6. Model of an AND gate

Figure 6 is a listing of the LISP function that models a two input AND gate. Basically, it is an implementation of the truth table for an AND gate. The first few statements obtain the value of the nets connected on input pins of the gate and store these values in variables *pin1-value* and *pin2-value*. The variables *pin1-value* and *pin2-value* are used in nested case statements to determine the value of the output of the AND gate. At the end of the model in the *list* statement, a resulting event for the net on the output pin of the gate is created. This event is scheduled to occur five time units in the future.

Resulting events generated by models are returned with their scheduled time set in relative form. The event queue manager will then change the event times to absolute times. The benefit gained from creating events whose scheduled times are in relative form is that delays in the events can be manipulated exterior to the model. If the expert system needs to extend the delay in an event, reverse computations are not necessary to calculate the original delay length.

Simulator

The third major component of the SWIM system is the simulator. Figure 7 contains an expanded view of the SWIM system architecture of Figure 1. In Figure 7, the simulator is expanded into its subcomponents. The simulator is a discrete-event simulator containing event and time handling utilities and a simulation clock. One utility, the event queue manager, maintains the event queue. Another utility, the time manager/sequencer, tracks the simulation time, generates trace information, and schedules event handling.

Figure 7. Expanded SWIM architecture

Simulation activity proceeds according to the following control algorithm:

- 1) The event queue manager passes all event(s) scheduled to occur next to the time manager.
- 2) The time manager updates the system clock, records these changes on the trace output, and schedules evaluation of all events with the evaluator.
- 3) The evaluator determines behavior of the models being excited, consults the expert system for any simulation advice, and passes any resulting events to the event queue manager.

Event Queue and Event Queue Manager

The event queue manager maintains the event queue. Events may represent net or memory transitions scheduled at a specific time or the re-ignition of a knowledge source at a specific time. During initial system set-up, the event queue manager constructs an event queue from an initial stimulus input file. During system operation, the event queue manager inserts all new events in the event queue when delivered by the evaluator.

The event queue is an array of 500⁴ lists treated as a circular queue, called a time wheel. Each list represents a time step, contains event(s) occurring at the same time, and will be accessed through its array index. An index into the array, called the CURRENT-TIME index, represents the current time. The array index following the CURRENT-TIME index represents one time step into the future. Because the array is circular, the array index preceding the CURRENT-TIME index represents 499 time steps into the future. Anything beyond 500 time steps is stored in an unordered list referred to as the "far list". The time wheel data structure was first designed by Ulrich and Suetsugu (1986).

During a simulation, the event queue manager manipulates the time wheel. When all the events of the current time step are processed, time is advanced by incrementing the CURRENT-TIME index until a non-empty event list is found. Each time the starting array index is reached, the far list is scanned once and any events occurring in the next 500 time steps are moved into the time wheel. Upon receiving new events, the event queue manager places any events scheduled for the next 500 time steps directly into the time wheel, while remaining events are stored in the far list.

⁴ The value 500 was chosen arbitrarily; this can be changed by modifying a program constant.

Time Manager.

The time manager updates the simulation clock, generates trace information, and schedules event evaluation with the evaluator. The event queue manager sends the time manager a list containing event(s) scheduled to occur next in the simulation. The time manager then updates the clock to the time of these scheduled events. Next, any events occurring on nets designated to be traced by the simulation window interface are recorded in a trace file. The trace information includes the net name, the transition type, and the transition time. In addition, if the event was influenced by the expert system, this is noted in the trace file. When completed with trace generation, the time manager passes the list of events to the evaluator for behavioral analysis.

Clock.

The clock is a globally accessible object which is used by the evaluator and expert system. The evaluator uses the clock time to calculate the time for future events. Expert system rules can also read the clock time. The clock is updated by the time manager based on the event queue and is the same as the most eminent event(s).

Evaluator

The forth major component of the SWIM system is the evaluator. Using current events specified by the time manager, the evaluator modifies the WDB and determines circuit behavior. Events reaching their scheduled time will either represent net transitions, memory device changes, or re-ignition of knowledge sources. If the event is a net transition, and the transition will change the world data base, then the evaluator changes the net's value; otherwise, net transitions

that will not cause a change in the world data base are ignored. During a simulation, the evaluator is the only module which modifies the world data base. Events calling for re-ignition of a knowledge source do not require that the knowledge source's trigger be met. These types of events can be created by a knowledge source to continue monitoring circuit behavior after an initial ignition.

When a net transition changes the world data base, the evaluator determines which parts and knowledge sources are affected. Parts with changing input pin values have their behavior analyzed first. Next, knowledge sources becoming ignited, according to their triggers, are started. The evaluator determines that all knowledge sources are completed by examining the "Ready" section of the blackboard (The blackboard will be discussed further in the section on the Expert System.). If a knowledge source surmises a behavior different from the simulator, it can manipulate previously created events or create new events by suggesting changes to the evaluator via the modification section of the blackboard. The evaluator uses these suggestions to update its list of resulting events. When all events for a time step have been processed the evaluator passes behavioral results to the event queue manager as future events.

Expert System

The fifth and final major component of the SWIM system is the expert system. The expert system is comprised of multiple expert systems, referred to as knowledge sources, and a blackboard (see Figure 7.). The blackboard is a data base for the knowledge sources. Each knowledge source contains a rule set, private working memory, and a suggestion buffer for communications to the evaluator. Working memory is used by a knowledge source to hold facts derived by its rules

during operation. In addition, there is a public memory where knowledge sources can store and modify facts. This public memory may serve as a channel for coordinating communications between different knowledge sources.

Knowledge Sources

Knowledge sources are expert systems defined to perform over a subcircuit or reduced domain (Hayes-Roth, Waterman, and Lenat 1983). A knowledge source in SWIM is defined using three parts. The first part describes a subcircuit pattern on which the knowledge source contains knowledge. The second part is a rule base or rule set applying to the subcircuit pattern. The third part, the ignition statement, defines events or starting conditions that will activate a knowledge source and start processing of its rule base using the inference engine.

The knowledge source description is constructed in a textual file using the TI Explorer's ZMACS window interface and the knowledge source's parts are applied at different times during a simulation. The rule base and ignition statement information will be used during simulation, while the subcircuit description will be used before a simulation is started to search for instances of this knowledge source. The subcircuit description is described using a domain definition statement. Two types of domain definitions can be made, TREE and SET.

TREE domain definitions define a subcircuit with connections between the parts. The TREE domain definition will cause the SWIM system to do a depth-first search to find subcircuits rooted with the same type of part as the domain definition pattern. Figure 8 illustrates a TREE domain definition for the circuit of Figure 4.


```

(def-domain '(TREE
              ((part or ref-or)
               ((part and ref-an1)
                a
                ((part not ref-n1) enable))
               ((part and ref-an2)
                enable
                b))))

```

Figure 8. Example of a TREE domain definition for circuit of Figure 4.

In a TREE domain definition, the first object in the list is a part. Succeeding objects are assumed to be connected to the inputs of this part in the same order as they are listed. If a succeeding object is another part, its output is assumed to be connected to the first part's input. A part in the domain definition pattern is designated by the list which starts with the operator *part*. The first argument of the *part* list is the part type and the second argument is its reference name. The reference name is used to refer to this object in the ignition statement and rule base for this knowledge source. Signals in the domain definition are defined as non-lists or atoms. The name used for the atom will be the reference name used for this signal in the ignition statement and rule base.

```

(def-domain '(SET
              (part outpad outpad1)
              (part outpad outpad1)
              (part outpad outpad2)
              (part outpad outpad3)
              (part outpad outpad4)))

```

Figure 9. Example of a SET type domain definition

SET domain definitions group many parts that may or may not be connected into a set. Figure 9 uses the SET domain definition in an example based on simultaneously switching bond pads explained in Chapter 3. This example domain definition will match the first five parts of type *outpad* found in the WDB.

Multiple TREE statements may be used to define a knowledge source domain. Only subcircuits which match all the TREE statements will be considered a match. Figure 10 shows an example of a multiple TREE domain definition which performs similarly to the domain definition of Figure 8.

```
(def-domain '(TREE
              ((part or ref-or1)
               (part and ref-an1)
               ((part and ref-an2)
                enable
                b))
              ((part and ref-an2)
               a
               ((part not ref-n1)
                a
                ((part not ref-n1) enable))))))
```

Figure 10. Multiple TREE type domain definition for circuit of Figure 4.

The ignition statement is a boolean statement that references parts and nets in its domain to define the conditions which might require expert help from a knowledge source. For instance, if a knowledge source was defined to be activated every time a net *s* becomes high, the statement would look like "(becomes (net 's) 'high)". During system initialization the ignition statement will be translated into an evaluable expression called a trigger. The trigger will use absolute references for all parts and nets instead of relative references. A trigger will be constructed for each instance of knowledge source type.

The knowledge source rule base is constructed with if-then rules modeled after the CLIPS language (Culbert 1987)⁵. Additional utilities were added to the rule language in SWIM for accessing the event queue, the world data base, and the LISP language. All rules reference parts, pins, and nets within their defined domain. These variable circuit references are bound to absolute entities in the circuit when the rules are used. This binding is similar to the matching of variables for function arguments in high level programming languages. Just as functions may be called multiple times with different arguments, the same knowledge source rules may be used by multiple instances of the knowledge source.

These rules are used in a forward chaining inference engine. The inference engine is modeled after the inference engine used in the expert system built with streams in Winston and Horn (1984). Rules are processed in the order in which they are defined. When a rule fires and produces a new fact(s), the rule list is rescanned starting from the beginning. If a complete scan through the rule list produces no new facts, the inference engine is stopped.

Suggestions to the evaluator result in event manipulation. Events associated with nets or knowledge sources can be changed, created, or destroyed. The most recently determined future events for this time slice are directly accessible to the knowledge source. These events can be modified or destroyed directly without going to the event queue. Other events that are in the event queue can only be modified by the event-queue manager. If the knowledge source finds that an event in the event-queue must be modified or destroyed, then messages for this purpose are passed from the knowledge source to the event queue manager via the evaluator. In addition, new events can be created by the knowledge source. These new

⁵ CLIPS is an expert system shell constructed by NASA and is written in the C language.

events are added to a list of resulting events. Thus, when the inference engine has been stopped, the knowledge source returns two items to the evaluator. The first is a set of resulting events and the second is a set of messages requesting modifications of event-queue events.

Figure 11 illustrates the format of the LISP structure that will hold a knowledge source's definition. The first field in the structure is the knowledge source's name. When an instance of a knowledge source is found, each will be named using this name field as part of its name. The remaining three fields (domain, ignitions-state, and rule-base) contain the three parts of the knowledge source definition described above (domain definition, ignition statement, and rule-base).

```
;; All references to circuit parts and nets are considered
;; relative to the domain definition for this knowledge source.
;; When this Knowledge Source is activated, actual circuit names
;; (nets, and parts) must be bound to these "relative circuit names."
;;
(defstruct ksdef
  "Knowledge Source Description"
  (name 'unknown)
  (domain nil)           ;Pattern domain of this KS
  (ignition-state nil)   ;Conditions for starting this KS
  (rule-base nil)       ;Rule base
)
```

Figure 11. Knowledge source definition record

Blackboard

The blackboard is a data base containing knowledge source information and information required for communications between the evaluator and the expert system. Working memory for each knowledge source, knowledge source status flags,

ignition state information, and resulting expert system suggestions are the types of information held in the blackboard.

SWIM's Dynamic Behavior

In this section SWIM's dynamic behavior is described. When a simulation session starts, several initialization procedures are performed to prepare the system for dynamic operation. First, the WDB, event queue, and model base are built from information in textual files. Second, knowledge sources are searched for in the WDB and their configuration is stored in the blackboard. In addition, when a knowledge source is found, the WDB is modified to indicate when a knowledge source's trigger should be evaluated. Recall, the trigger is a translated ignition statement.

When a knowledge source is found, the blackboard and the WDB are modified to record the knowledge source's presence. For each instance of a knowledge source found, the relative part and net names used in the knowledge source's domain definition are mapped to absolute names. These mappings are saved in a binding list which is stored in the blackboard. When a knowledge source is started by the evaluator during simulation, these bindings will be used to map rule references to their WDB parts and nets. Pin structures are also added in the WDB when a knowledge source is found so that the evaluator recognizes nets which can trigger a knowledge source. When a net changes and a pin belonging to a knowledge source is found on the net, the appropriate knowledge source trigger statement is evaluated.

Figure 12 illustrates the LISP structures containing information for each instance of a knowledge source. When an instance of a knowledge source is recog-

nized, it is cataloged in the blackboard by creating entries for:

- 1) a relative-to-absolute mapping (called *alist* in Figure 12.),
- 2) a trigger,
- 3) a working memory area (called *workmem* in Figure 12.),
- 4) a suggestion area, and
- 5) a status word.

The working memory area, suggestion area, and status word are used during the inference operation on a knowledge source. Most of the parts of Figure 12 have already been explained except for the *eval-bit*. The *eval-bit* is used for book-keeping during the evaluator's operation to prevent multiple evaluations of the same part due to multiple pin changes on the part.

```
(defstruct ksinst
  "Knowledge Source Instance Description"
  (name 'unknown)
  (ksdef nil)      ;Points to the definition for this knowledge source.
  (alist nil)      ;Association list for binding parts and pins in rule-base
                  ;of KS definition to their actual part in WDB.
  (trigger nil)    ;A boolean statement which is evaluated to determine
                  ;ignition of this KS instance.
  (eval-bit nil)   ;A flag to show pending evaluation of this KS.
                  ; T--> Pending  NIL--> Not pending
  (event-queue nil) ;Events created by this KS. This list is
                  ;maintained by the Evaluator. Events will
                  ;exist both in this list and the system
                  ;event-queue. Each event in this list
                  ;points to the KS through its creator slot.
  (workmem nil)    ;Working memory for this KS.
  (suggestions nil)
  (status nil)
)
```

Figure 12. Knowledge source instantiation information.

```

FOR all events in the current time slice:
  IF this event represents a net change (non-impotent event.)
  THEN Change the net's value in WDB
    Find parts which should be evaluated on this net
    [Save these in list if not already there: eval-part-list]
    Find knowledge sources which should be visited
    [Save these in list if not already there: visit-KS-list]
  END-THEN
END-FOR

FOR all parts in list eval-part-list
  Evaluate the part's behavior
  [Save resulting events in list: Resulting-events]
END-FOR

FOR all Knowledge sources in visit-KS-list
  Evaluate the trigger statement for this KS instance
  [Save Knowledge sources which were triggered in list: eval-KS-list]
END-FOR

FOR all Knowledge sources which were triggered (held in eval-KS-list)
  Evaluate the knowledge sources rule base.
  [Save resulting events in list: Resulting-events]
END-FOR

RETURN Resulting-events

```

Figure 13. Evaluator pseudo code

Once system initialization is complete, the evaluator does most of the simulator work as discussed in a previous section. A more detailed description of the evaluator's operation is shown using pseudo code in Figure 13. Four consecutive FOR-loops do the event evaluation. The first FOR loop determines the parts in the WDB which are actually stimulated by events which have become active. Parts with multiple input pins changing are only evaluated once. The second FOR loop does the part model evaluation. All resulting events are saved in case they

need to be further modified by the expert system. The third FOR loop determines which knowledge sources may need to be called. During system initialization, pins were added in the WDB on nets which were referenced in knowledge source ignition statements. When one of these nets changes, the knowledge source pin alerts the evaluator that a trigger statement should be evaluated. The fourth FOR loop activates knowledge sources which were triggered. In this loop, the expert system's inference engine is started using the knowledge source's rules and facts. Any resulting actions suggested by the expert system will be applied to the events generated in the previous FOR loops.

SWIM User Interface

Simulation of a digital circuit is controlled through the Explorer's interactive windowing system. From a special SWIM simulation window, a designer can:

- 1) load a circuit description from a file into the simulator.
- 2) construct simulation stimulus for a circuit.
- 3) initialize a circuit's state by setting nets and memory element values.
- 4) collect simulation results in a file.
- 5) manipulate the expert system's knowledge-base.
- 6) start, stop, or step through a simulation.
- 7) switch into a ZMACS text editor window to create or modify a circuit description.
- 8) save a simulation session for later reuse.

<pre> 300 Event{KS-Global-Net,300,X} 400 Event{A,0,1} Event{B,0,1} Event{C,0,1} Event{D,0,1} Event{S0,0,1} Event{S1,0,1} 450 Event{S0,0,X} Event{S1,0,X} 500 Event{S0,0,1} Event{S1,0,1} 550 Event{S0,0,X} 600 Event{S0,0,1} 650 Event{S1,0,X} 700 Event{S1,0,1} =====Current Transitions 300 Event{D,0,1} Event{PADOFF,0,1} "Parts needing evaluation (and their events) " ((Part{CMP87,INPAD} (Event{PADOFF,0,1})) (Part{CMP69,INPAD} (Event{D,0,1}))) "List of KS's which will be started: " NIL =====New Transitions and Deletions Add: Event{DD,8,1} Event{POFF,0,1} Delete: -NONE- =====Event Queue 300 Event{POFF,300,1} Event{DD,300,1} Event{KS-Global-Net,300,X} 400 Event{A,0,1} Event{B,0,1} Event{C,0,1} Event{D,0,1} Event{S0,0,1} Event{S1,0,1} 450 Event{S0,0,X} Event{S1,0,X} 500 Event{S0,0,1} Event{S1,0,1} 550 Event{S0,0,X} 600 Event{S0,0,1} 650 Event{S1,0,X} 700 Event{S1,0,1} =====Current Transitions 300 Event{POFF,300,1} Event{DD,300,1} Event{KS-Global-Net,300,X} "Parts needing evaluation (and their events) " ((Part{CMP93,AND3} (Event{DD,300,1})) (Part{CMP75,AND3} (Event{DD,300,1})) (Part{CMP96,AND3} (Event{POFF,300,1}))) "List of KS's which will be started: " (KS_Inst(RAMCNT-REFRESH-SET,KS_Def(RAMCNT-REFRESH))) (RULE RAM-NOT-REFRESHED USES LISP EXPRESSION (CREATE-EVENT (PART-PIN Part{CMP16,RAMCNT} 11) :TIME 1 :VALUE 'X) AND RETURNS Event{CELLRW,1,X}) (RULE RAM-NOT-REFRESHED SAYS FORGET 2FORGET2 (INITIAL-FACT)) (RULE RAM-NOT-REFRESHED SAYS DONE) =====New Transitions and Deletions Add: Event{FA1,5,1} Event{AN4,5,1} Event{FA4,5,0} Event{CELLRW,1,X} Delete: -NONE- =====Event Queue 309 Event{CELLRW,309,X} 313 Event{FA4,313,0} Event{AN4,313,1} Event{FA1,313,1} **MORE** SWIM </pre>	<p>Signal names</p> <pre> F W01 Y F 01 RTTY X 0 010 1 NN LOUX U D DDN N FII LOOX M0101 23 A 01DDI 1WEDD EDDMU FSSYY YY ABCDP ISSAAD DRRCC CDDFM NRRRR RR F W01 Y F 01 RTTY X 0 010 1 NN LOUX U D DDN N FII LOOX M0101 23 A 01DDI 1WEDD EDDMU FSSYY YY ABCDP ISSAAD DRRCC CDDFM NRRRR RR 0 XXXXX XXXXX XXXXX XXXXX XXXXX XX 10 0100 1111 100XX XXXXX XXXXX XX 31 0100 1111 100XX XXXXX XXXXX XX 34 0100 1111 100XX XXXXX XXXXX XX 100 0100 1111 101XX XXXXX XXXXX XX 110 0100 1111 100XX XXXXX XXXXX XX 122 0100 1111 10011 XXXXX XXXXX XX 128 0100 1111 10011 XXXXX XXXXX 01 200 0100 1111 11011 XXXXX XXXXX 01 212 0100 1111 11011 XXXXX XXXXX 01 218 0100 1111 11011 XXXXX XXXXX 01 300 0111 1111 11011 XXXXX XXXXX 01 </pre> <p>Traces</p>
<p align="center">UNT SWIM Simulation System</p> <p>SWIM: (EVENT-QUEUE PART-LIST MODEL-LIST KS-DEF-LIST SWIM-FRAME)</p>	

07/05/89 11:25:37PM BENAVIDES

USER:

Run

+ LEWIS: BENAVIDES.SWIM; SIMTRACE.TEXT#61 3496

Figure 14. SWIM simulator window

Figure 14 illustrates the SWIM simulation window frame. There are three subwindows to the window frame. The first subwindow, labeled **SWIM** in its lower left corner, is used for displaying the simulator messages, expert system traces, and event-queue contents. The second subwindow, labeled **Traces** in its lower left corner, is used to display the value of nets (signals) in the circuit (WDB). Each column corresponds to one net. When any displayed net changes, a new row containing all the signal values is printed in the subwindow for the time when it changed. This is called a print on change listing. The third subwindow, labeled **Signal names** in its top left corner, is used to display the name of the nets being traced in each column of the the **Traces** subwindow.

Simulation results are also stored in files. The signal values displayed in the **Traces** subwindow are also stored in a file. Also, as each event of the simulation is consumed, it is logged in an event trace file. This logging preserves knowledge of impotent and overridden events. Impotent events are those which try to change nets which are already at the desired value. Overridden events are those which the expert system advised the evaluator to ignore and were disregarded. Overridden events also include those which no longer exist in their original form because they were modified by the expert system.

Summary

In this chapter, the SWIM system has been described. The five components of the system have been defined and their purpose has been explained. In summary:

- o The WDB holds the circuit's information.

- o The model base contains the part behavior descriptions used for simulation.
- o The simulator controls the normal model evaluation and event handling.
- o The expert system determines non standard circuit behavior.
- o And, the evaluator integrates the expert systems and the simulator operation.

Finally, an explanation of SWIM's dynamic behavior and user interface was given.

CHAPTER III

SYSTEM EXAMPLE

This chapter describes the operation of the SWIM system by discussing an example circuit and its simulation trace. The circuit does not perform any meaningful operation, but is instead designed solely to illustrate the interaction between SWIM system components. The chapter begins with an explanation of the example circuit. Three knowledge sources (KS) which will be active during the simulation are also described. After the circuit and KS descriptions, portions of the circuit's simulation trace are examined and explained.

The Circuit

The top level view of the example circuit used in this chapter is shown in Figure 15a. The example circuit contains two independent subcircuits. The first subcircuit, Figure 15b, is a collection of simple logic gates which perform as two different multiplexing circuits. The second subcircuit, Figure 15c, is a four by two dynamic random access memory (DRAM) and its refresh circuit. More complex primitives containing internal memory are used in the second subcircuit.

All signals entering and leaving the example circuit pass through input and output pads. This allows modeling of the circuit as a VLSI chip. The purpose of the pads on a VLSI chip is to translate external currents and voltages levels to the chip's internal levels. For this example, a pad can simply be modeled as a buffer which passes a logic value from its input to its output after a specified delay.

Figure 15. (a) Example Circuit; top level view

Figure 15. (b) Example Circuit; first subcircuit

Figure 15. (c) Example Circuit; second subcircuit

First Subcircuit

The first subcircuit is constructed with two multiplexers, each of which has four inputs and one output (see Figure 15b). Inputs for multiplexer-X are the signals A, B, C, and D. The output from multiplexer-X is the signal MUXY. Inputs for the multiplexer-Y are the signals PADOFF, B, C, and D. The output from multiplexer-Y is the signal FMUXY. To select from the four different inputs on both multiplexers, the signals S0 and S1 are used. The two multiplexers perform similarly, but differ in their select line configuration. The truth table in Figure 16 defines the input signal whose value is passed to the output for different values on the select lines.

Input values		Input signal sent to Output	
S1	S0	MUXY	FMUXY
0	0	A	D
0	1	B	C
1	0	C	B
1	1	D	PADOFF

Figure 16. Input signal whose value is used on multiplexer outputs.

Second Subcircuit

The second subcircuit (DRAM) is more complex than the first one and contains three types of primitive (see Figure 15c). These primitives are the RAMCELL, the ROWSEL, and the RAMCNT. The RAMCELL (RAM cell) is a one bit dynamic RAM. The RAMCNT (RAM controller) controls the operation and refresh of the RAMCELL primitives. The ROWSEL (row selector) is a primitive which selects one of its four outputs to be a high value depending on its two inputs. All three primitive types are used to construct a DRAM circuit containing

four words of two bits.

The DRAM circuit is constructed in the following fashion. Eight RAMCELL primitives are used to construct a four by two RAM array. All inputs in the first RAM column are tied together to form one bit of an input data bus (signal CDIN0). Similarly, all the inputs in the second RAM column are tied together to form the second bit in the input data bus (signal CDIN1). Outputs of each RAMCELL are also tied together in an analogous fashion as the RAMCELL inputs to form an output data bus (signals DOUT0 and DOUT1). RAMCELL primitives on the same row have their select pins tied to the same signal which is controlled by the ROWSEL primitive. The ROWSEL operates according to the truth table in Figure 17.

Input values		Output values			
RS0	RS1	RY0	RY1	RY2	RY3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Figure 17. ROWSEL truth table.

Figure 18 illustrates the behavior of the RAMCELL using a truth table. The RAMCELL primitive performs in the following fashion. The *Select* (see Figure 18) effectively enables or disables the RAMCELL. When the *Select* pin is at a logic zero level, the output pin, *Dout*, has a value of high impedance. A pin with a high impedance value can be thought of as not being connected to the signal; the pin has no driving effect on the value of the signal. When the *Select* pin is at a logic one level, it is enabled. When enabled, the RAMCELL will perform according to the value on its *RW* pin. If the *RW* pin is stable at a zero or one value, the con-

tents of the RAMCELL are displayed on *Dout* pin. Whenever the RAMCELL is enabled and a rising edge occurs on the *RW* pin, the value stored in the RAMCELL will be changed to the value on the *Din* pin. A more detailed description of the RAMCELL's behavior can be found in the LISP model used in this example simulation (see Appendix).

Inputs			Memory Contents	Output Dout	VALUES	
Select	RW	Din				
0	?	?	SAB	Z	0	Logic 0
1	0	?	SAB	Contents	1	Logic 1
1	1	?	SAB	Contents	?	Don't care
1	0->1	0	0	0	Z	High impedance
1	0->1	1	1	1	SAB	Same as before
					Contents	Memory contents

Figure 18. RAMCELL operation table.

The DRAM circuit's operation is controlled by the RAMCNT primitive. This primitive changes the entire DRAM between one of three states: read, write, or refresh. The signal *RW* is used to define a read or write state. The read state is entered whenever the *RW* signal has the logic value low. The write state is entered when the *RW* line experiences a rising edge (transitions from 0 to 1). When the circuit is in a read or write state, a word of the memory is selected with the address signals *ADD0* and *ADD1*. When reading from the DRAM, data is taken from the data output signals *DOUT0* and *DOUT1*. When writing to the DRAM, data is provided via the data input signals *DIN0* and *DIN1*. In the read or write state, the RAMCNT primitive passes values on the signals *ADD0*, *ADD1*, *DIN0*, *DIN1*, and *RW* through to the signals *RS0*, *RS1*, *CDIN0*, *CDIN1*, and *CELLRW*, respectively.

When the DRAM is in a refresh state, the RAMCNT primitive does not pass values. Instead, the DRAM cycles through each memory word of the RAM array, first reading, then writing values back. The DRAM circuit enters a refresh state

whenever the REF signal experiences a rising edge (transitions from 0 to 1). If the DRAM is currently in a refresh state and the signal REF experiences a rising edge, the edge is ignored.

Knowledge Sources

Three knowledge sources are used in this example simulation. The first KS is an expert on the behavior of simultaneously switching output pads. This KS will adjust simulation results if multiple pads switch simultaneously to emulate power supply problems. The second KS is an expert on the behavior of reconvergent signals in a four input multiplexer. This KS will increase simulation accuracy by using inter-circuit signal relationships to evaluate the behavior of a multiplexer sub-circuit. Finally, the third KS is an expert on DRAM refresh operations. If a DRAM is not refreshed in time to preserve its memory values, this KS will change its contents to unknowns.

Switching-pads KS

The first knowledge source is designed to detect simultaneously switching outputs of an integrated circuit (IC). In advanced CMOS IC's, multiple switching outputs can cause noise in the resulting output waveforms (Texas Instruments 1987). For a short time, the outputs should be considered unknown and thus unusable. This knowledge source, called switching-pads, is designed to detect simultaneous switching of four or more output pins in the example circuit. In the following paragraphs, this KS will be explained.

The pattern domain definition for the switching-pads knowledge source is described in Figure 19. The domain definition fulfills two purposes; it defines a subcircuit upon which the knowledge source has knowledge and assigns relative

references to the subcircuit. The domain definition of Figure 19 uses the SET keyword explained in Chapter 2. The SET keyword specifies that the set of objects following may or may not be connected. In this case, the set of objects in this domain definition are seven parts of the type *outpad*. References established in the domain definition will also apply in other parts of the KS definition (ignition statement and rule base).

```
(def-domain '(SET (part outpad outpad1)
                  (part outpad outpad2)
                  (part outpad outpad3)
                  (part outpad outpad4)
                  (part outpad outpad5)
                  (part outpad outpad6)
                  (part outpad outpad7))))
```

Figure 19. Domain definition for switching-pads KS.

```
(def-ignition '(or (netchange (part-pin outpad1 2) ? 1)
                   (netchange (part-pin outpad2 2) ? 1)
                   (netchange (part-pin outpad3 2) ? 1)
                   (netchange (part-pin outpad4 2) ? 1)
                   (netchange (part-pin outpad5 2) ? 1)
                   (netchange (part-pin outpad6 2) ? 1)
                   (netchange (part-pin outpad7 2) ? 1)))
```

Figure 20. Ignition statement for switching-pads KS.

The ignition statement for this KS is shown in Figure 20. The knowledge source is defined to be triggered when any of the seven parts (*outpad1*, *outpad2*, ... *outpad7*) has its second pin's signal go high. In other words, this ignition statement is edge sensitive. Only a transitioning signal will cause it to be true. In the ignition statement, the operator *netchange* does this checking. *Netchange* checks its first argument (a net or pin) to determine if its value has transitioned from the

value supplied as the second argument to the value supplied as the third argument. In Figure 20, the logic value ? is a wild card for any logic level.

Figure 21 illustrates the rule base for the knowledge source *switching-pads*. There are nine rules in the set. The first rule, called *start*, initializes the knowledge source. When any knowledge source is initiated, an initial fact, *initial-fact*, is placed in its working memory. The initial fact activates the first rule in this rule base, which counts the number of simultaneously switching outputs using the *dolist* loop. A *dolist* takes each item in a list and binds it to a loop variable for one pass through the body of the loop. In this case, the loop variable is *?sig* and the values in the list are pin 2 of part *outpad1*, pin 2 of part *outpad2*, etc. The *dolist* also checks to see if any of the outputs are going high or will go high in the near future by using the *went* construct. As the first rule is counting, facts are created specifying which outpad parts are switching to high. The first rule also creates an additional fact, *counted*, in order to activate future rules.

When a rule's antecedent is true, it is said to be firing. A rule which fires will generate actions which may result in facts. The facts are placed in working memory. When the fact *counted* is placed in working memory after rule one is completed, it will cause the second rule to fire. Rule two, called *determine-delays*, determines if a simultaneous switching state exists and, if so, an appropriate delay factor is computed. Rules three through nine are all similar; each determines if one of the seven different outpads is switching and the appropriate action that should be taken. For example in rule three, if *outpad1* is switching, then an event is created to model noise generated by the switching. Following the period of noise, the signal should stabilize and change as planned. A second event is created to show the eventual transition of *outpad1*'s second pin to high. The last action carried out by the rule is to print a message noting the behavior discovered.

```

;1. Count number switching and assert facts for each
(defrule start
  (if (initial-fact))
  (then (bind ?count 0)
        (assert (counted))
        (forget (initial-fact))
        (dolist ?sig '((part-pin outpad1 2)
                        (part-pin outpad2 2)
                        (part-pin outpad3 2)
                        (part-pin outpad4 2)
                        (part-pin outpad5 2)
                        (part-pin outpad6 2)
                        (part-pin outpad7 2))
          (if (went 'high ?sig :within 5 'ns)
              (bind ?count (+ ?count 1))
              (assert (?sig switching to high)))))))

;2. Determine the delay factor
(defrule determine-delays
  (if (counted)
      (test (>= ?count 4)))
  (then (bind ?delay-factor (+ 1 (* (- ?count 2) .10)))
        (assert (modify events)))))

;3. Take action:
(defrule check-switch-outpad1
  (if (modify event)
      ((part-pin outpad1 2) switching to high))
  (then (create-event (part-pin outpad1 2)
                      :time (get-event-field (part-pin outpad1 2)
                                                :time)
                      :value 'X)
        (modify-event (part-pin outpad1 2)
                      :time (* ?delay-factor
                               (get-event-field (part-pin outpad1 2)
                                                  :time)))
        (print-msg "Simultaneous switching behavior found..."
                    (part-pin outpad1 2))))

;4. - 9. are similar to 3. except that outpad1 is exchanged for
;outpad2, outpad3, outpad4, outpad5, outpad6, and outpad7, respectively.

```

Figure 21. Rule base for switching-pads KS.

Multiplexer KS

The second knowledge source is designed to restore accuracy to the simulation of a four input multiplexer. When circuits are modeled at a boolean level and a signal diverges and reconverges, behavioral information is lost (Miczo 1986). This is because gate level simulators do not use inter-circuit signal relationships to evaluate the behavior of circuits. Gate level simulators evaluate a circuit's behavior based only on the behavior of its parts. The three most common parts in a gate level simulator are the NOT, AND, and OR gates; their behavior is shown in Figure 22 (Valid 1985, Mentor 1986, GenRad 1985).

Inputs		AND	OR
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
0	X	0	X
1	X	X	1
X	0	0	X
X	1	X	1
X	X	X	X

Figure 22. (a) Behavior of AND and OR gates

Input	NOT
0	1
1	0
X	X

Figure 22. (b) Behavior of NOT gate

In order to illustrate the kind of problem this knowledge source corrects, consider the circuit in Figure 23a. The behavior of the circuit in Figure 23a is simulated in Figure 23b. In the first line of Figure 23b, the equation for the circuit is shown. In the second line, the simulation starts with initialization of the circuit.

Initialization is accomplished by allowing the circuit to settle after applying the value 1 to signals *A*, *B*, and *E*. In the third line, the value on signal *E* changes and causes two gates, the *NOT* and the right *AND*, to be evaluated by the simulator. Evaluation of both gates results in causing their output pins to become unknown (*x*) as shown in the fourth line. In the fifth and sixth lines, the left *AND* gate and the *OR* gate are evaluated when their inputs change resulting in the value of signal *Y* becoming unknown (*x*). The performance of the simulator results in determining an incorrect result for signal *Y*.

$Y = (A \text{ AND } (\text{NOT } E))$	$\text{OR } (B \text{ AND } E)$	-1- Equation for circuit of 23a.
$Y = (1 \text{ AND } (\text{NOT } 1))$	$\text{OR } (1 \text{ AND } 1)$	-2- All signals initialized to 1
$Y = (1 \text{ AND } (\text{NOT } x))$	$\text{OR } (1 \text{ AND } x)$	-3- <i>E</i> changes to <i>x</i>
$Y = (1 \text{ AND } (x))$	$\text{OR } (x)$	-4- NOT and right AND gate evaluated
$Y = (x)$	$\text{OR } (x)$	-5- left AND gate evaluated
$Y = (x)$	x	-6- OR gate evaluated

Figure 23. (a) *Top* Simple circuit with reconvergent fanout,
(b) *Bottom*, Sample simulation.

If the circuit of Figure 23a is evaluated as a boolean function, much the same way as a theorem prover operates, correct results for the simulation are obtained. The value for Y can be derived with the equation below:

$$Y = (A \text{ AND } (\text{NOT } E)) \text{ OR } (B \text{ AND } E)$$

Substituting 1 for signals A and B reduces the first equation to the equation shown below:

$$Y = (\text{NOT } E) \text{ OR } E$$

This second equation is a tautology. Signal Y in the circuit should have had a final value of 1. The cause of incorrect behavior in the simulator can be traced to the split of signal E at the NOT gate. The simulator operates incorrectly when the signal fragments of E are reunited at the OR gate because the fragments are treated as two independent signals. This incorrect behavior by the simulator is known as the reconvergent fanout problem (Miczo 1986).

In the example simulation of Figure 15, a slightly larger subcircuit which exhibits the reconvergent fanout problem (shown in Figure 23) and a knowledge source which corrects the problem will be illustrated. The subcircuit exhibiting the reconvergent fanout problem is a four input multiplexer. Figure 24 shows the domain definition for this four input multiplexer KS. This domain definition is of the *TREE* type described in the Chapter 2. It uses part connectivity to describe a circuit pattern. In the example circuit of Figure 15b, the domain definition of Figure 24 matches multiplexer-X, but does not match multiplexer-Y. The multiplexer KS will correct the reconvergent fanout problem.

```

(def-domain '(TREE ((part or4 m-or)
                    ((part and3 m-an0)
                     in1
                     ((part inv m-nt0) sel0)
                     ((part inv m-nt1) sel1))
                    ((part and3 m-an1)
                     in2
                     sel0
                     ((part inv m-nt1) sel1))
                    ((part and3 m-an2)
                     in3
                     ((part inv m-nt0) sel0)
                     sel1)
                    ((part and3 m-an3)
                     in4
                     sel0
                     sel1))))

```

Figure 24. Domain definition for multiplexer KS.

To correct the reconvergent fanout problem, the multiplexer KS will perform as a simplified theorem prover. It will correct the simulator's derived behavior by adjusting its results. To determine the correct results for the multiplexer subcircuit, a truth table describing the behavior of a four input multiplexer can be constructed. Using all possible values (1, 0, and X^1) on the input signals of the four input multiplexer subcircuit results in a truth table having 729 rows. In this fully expanded truth table, 37 out of 729 rows would be wrong. In Figure 25, the 37 wrong rows are reduced to 5 rows using wild cards for groups of rows. The right-most column in each row describes how many rows of the fully expanded truth table are represented in that row. In Figure 25, one column describes the simulator's determined behavior and another column describes the knowledge source's adjustment to that behavior.

¹ X represents unknown.

Inputs						Simulator's	KS's	rows
in1	in2	in3	in4	sel0	sel1	Y value	Y value	represented
1	1	1	1	X	X	X	1	1
1	1	?	?	X	0	X	1	9
?	?	1	1	X	1	X	1	9
1	?	1	?	0	X	X	1	9
?	1	?	1	1	X	X	1	9

? - Represents a wild card for any value (0,1, or X)

Figure 25. Simulator and KS evaluated behavior for multiplexer circuit.

The primitives in this knowledge source's domain are level sensitive. If at any time an input on these parts changes, a new event is created for the part's output regardless of its current value. This is not the case for edge sensitive parts. Edge sensitive parts only react to a specific transition from one logic level to a different level. The difference between level and edge sensitivity is subtle. This KS is level sensitive in order to deal with the level sensitive primitives in its domain. The level sensitive ignition statement for this KS is shown in Figure 26. If at any time either of the select lines (*sel0* or *sel1*) becomes X, this KS will be triggered.

```
(def-ignition '(or (equal (signal sel0) X)
                     (equal (signal sel1) X)))
```

Figure 26. Domain definition for multiplexer KS.

Figure 27 shows the rule base for the multiplexer KS. There are six rules in the set. The first rule queries the WDB to find the values of the six inputs to the multiplexer. The second rule fires if both the select lines, *sel0* and *sel1*, are unknown and all other inputs are 1. The second rule corresponds to the first row of the truth table in Figure 25. The second rule will correct the value of signal *Y* and prevent any other incidental events generated by an X on the select lines from

propagating through the KS's domain. The last four rules in the rule base perform similar to the second rule, but correspond to the remaining four rows of the truth table in Figure 25.

The SWIM inference engine processes rules in the order encountered. Since (*initial-fact*) is always inserted into working memory, the first rule will always fire when the KS is started. After the first rule has been fired, any other rule that fires will prevent succeeding rules from firing by removing the fact (*not done*) from the working memory.

DRAM Refresh KS

The third knowledge source is designed to monitor the refresh of a DRAM. If a DRAM is not refreshed within a defined period, the memory's voltage levels deteriorate and can not be sensed reliably. This is because each memory bit is basically a capacitor in which the charge can leak away over time (Miczo 1986). To emulate this type of behavior, this knowledge source will change the value of the example circuit's memory cells if a refresh is not performed within 200 ns.

The three parts of this KS (domain definition, ignition statement, and rule base) are shown in Figure 28. Since the RAMCNT primitive contains and controls the DRAM's refresh circuitry, this KS will be closely linked to the RAMCNT. The domain definition for this circuit is constructed with a *SET* type domain definition. The only part in the domain definition is the RAMCNT primitive. The ignition statement will become true when the RAMCNT is stimulated to start a DRAM refresh by the signal *RREF* on pin 6.

```

(defrule start ;Get the values
  (if (initial-fact))
  (then (sel0 is (net-value sel0))
        (sel1 is (net-value sel1))
        (in1 is (net-value in1))
        (in2 is (net-value in2))
        (in3 is (net-value in3))
        (in4 is (net-value in4))
        (forget (initial-fact))
        (assert (not done)))))

(defrule both-unknown
  (if (not done)
      (sel0 is X)
      (sel1 is X)
      (in1 is 1)
      (in2 is 1)
      (in3 is 1)
      (in4 is 1))
  (then (create-event (part-pin m-or 5) ;Output pin
                    :time 10 ;Time through AND stage and OR gate
                    :value 'X)
        (zap-events (part-pin m-or 2))
        (zap-events (part-pin m-or 3))
        (zap-events (part-pin m-or 4))
        (zap-events (part-pin m-nt0 2))
        (zap-events (part-pin m-nt1 2))
        (forget (not done)))))

(defrule sel0-unknown-sel1-0
  (if (not done)
      (sel0 is X)
      (sel1 is 0)
      (in1 is 1)
      (in2 is 1))
  (then (create-event (part-pin m-or 5) ;Output pin
                    :time 10 ;Time through AND stage and OR gate
                    :value 'X)
        (zap-events (part-pin m-or 2))
        (zap-events (part-pin m-or 4))
        (zap-events (part-pin m-nt0 2))
        (forget (not done)))))

```

Figure 27. Rule base for multiplexer KS (first part)

```

(defrule sel0-unknown-sel1-1
  (if (not done)
    (sel0 is X)
    (sel1 is 1)
    (in3 is 1)
    (in4 is 1))
  (then (create-event (part-pin m-or 5) ;Output pin
    :time 10 ;Time through AND stage and OR gate
    :value 'X)
    (zap-events (part-pin m-or 2))
    (zap-events (part-pin m-or 4))
    (zap-events (part-pin m-nt0 2))
    (forget (not done))))

(defrule sel1-unknown-sel0-0
  (if (not done)
    (sel1 is X)
    (sel0 is 0)
    (in1 is 1)
    (in3 is 1))
  (then (create-event (part-pin m-or 5) ;Output pin
    :time 10 ;Time through AND stage and OR gate
    :value 'X)
    (zap-events (part-pin m-or 3))
    (zap-events (part-pin m-or 4))
    (zap-events (part-pin m-nt1 2))
    (forget (not done))))

(defrule sel1-unknown-sel0-1
  (if (not done)
    (sel1 is X)
    (sel0 is 1)
    (in2 is 1)
    (in4 is 1))
  (then (create-event (part-pin m-or 5) ;Output pin
    :time 10 ;Time through AND stage and OR gate
    :value 'X)
    (zap-events (part-pin m-or 3))
    (zap-events (part-pin m-or 4))
    (zap-events (part-pin m-nt1 2))
    (forget (not done))))

```

Figure 27. (continue) Rule base for multiplexer KS

```

;; 1. Domain definition
(def-domain '(SET (part ramcnt ram_control)))

;; 2. Edge sensitive ignition statement
(def-ignition '(netchange (part-pin ram_control 6) ? 1)) ;Refresh pin changes?

;; 3. Rule base
(defrule start      ;Get the values
  (if (initial-fact)
    (went 1 (part-pin ram_control 6) :within 200))
  (then (forget (initial-fact))
        (zap-events :me)
        (create-event :me :time (+ (net-change-time (part-pin ram_control 6)
                                                    200))))
    (assert (done))))
(defrule RAM-not-refreshed
  (if (initial-fact))
  (then (create-event (part-pin ram_control 11) ;R/W line
                     :time 1 :value 'X)
        (forget (initial-fact))
        (assert (done))))

```

Figure 28. DRAM refresh KS.

There are two conditions which might cause this KS to be initiated. The first condition is when the RAMCNT starts a refresh operation on the DRAM. The second condition is when the KS reactivates itself with a special event to check on the state of the refresh operations. This reactivation ability was not demonstrated in either of the previous KS's. Events started by a KS for reactivation of the KS do not have to satisfy the KS's ignition statement.

The rule base for this knowledge source is shown in Figure 28. There are two rules in the set. The first rule fires if the signal on pin 6 of the RAMCNT, *RREF*, has caused a DRAM refresh. Three actions which result from the first rule. The first action is to prevent the second rule from firing. The second action is to destroy any pending events scheduled to restart this KS; these pending events will

have been started to verify that the DRAM was refreshed. The third action is to schedule a reactivation of this knowledge source at 200 ns from the time of the last DRAM refresh to verify a new refresh will be done.

The second rule of this KS fires if the first does not. This is because the rules are processed in their listed order. If the first rule does not fire, the KS has been started by a self created re-ignition event. This means the DRAM has not been refreshed. If it had been refreshed, the re-ignition event would have been stopped by the first rule. The action which results from the second rule is to create an event which will cause all the bits of the DRAM to become unknown. This is accomplished by setting the *CELLRW* signal to X (see Figure 15c).

This KS is over simplified in order to demonstrate the applicability of the KS concept. For example, if the *RREF* signal goes through a rising edge while the DRAM is in a refresh state, the KS will think that this was the time of the DRAM's last refresh. In order to prevent the KS from being fooled by the *RREF* signal changing multiple times, a smarter KS would follow the *RAMCNT*'s behavior more closely to determine if it is in a refresh state.

Traces

In this section, five excerpts from a simulation trace will be illustrated and explained. (Benavides 1989) contains the entire simulation trace from which these excerpts are taken. The first trace excerpt shows the initialization of the simulator for the example circuit of Figure 15. The second trace shows a normal evaluation of gate level models. Excerpts of the next three traces show operations of the knowledge sources described in the previous section.

Initialization Trace

The simulation trace in Figure 29 illustrates the initialization steps performed prior to simulation. The command (*load-startup-paper*) loads the circuit into the WDB, the simulation models into the model base, the initial events into the event-queue, and the KS definitions into memory. The command (*swim ...*) starts the simulator. The simulator then asks the user several questions to determine the level of details required for "trace information."

The simulator begins operation by searching for instances of each defined KS. The first search is for the multiplexer KS (called MUX-KS in trace). Because the domain definition for the multiplexer KS was defined with a *TREE* type pattern definition, a depth-first comparison of subcircuit trees is done. Only subcircuits rooted with the same type of part as the domain definition are tried. Two possibilities for this comparison exist in our example circuit, CMP78 and CMP74; both are *OR4* type parts. Only the subcircuit rooted at CMP78 is found to match. The list below the heading *RESULTING-BINDINGS* shows which parts and nets in the subcircuit match the KS's relative names. In a similar fashion, searches are performed for instances of the DRAM refresh KS and the switching-pads KS (called RAMCNT-REFRESH and PAD-SWITCH in trace).

After finding a KS, two tasks must be performed by the simulator before the KS can be incorporated into the simulation. The first task is to translate the ignition statement into an evaluable trigger that will be used at simulation run time to determine if a KS inference session should be started. The second task is to insert pseudo pins in the WDB for signals that are part of the trigger. These pseudo pins link the KS to the WDB and are used to determine when the evaluator will check a KS's trigger. The trace of both tasks are shown in Figure 29.

```
> (load-startup-paper)
; Loading LEWIS: BENAVIDES.SWIM; SWDB.LISP#> into package SWIM
; Loading LEWIS: BENAVIDES.SWIM; SWDB_MODELS.LISP#> into package SWIM
; Loading LEWIS: BENAVIDES.SWIM; SWDB_EVENTS.LISP#> into package SWIM
; Loading LEWIS: BENAVIDES.SWIM; SWDB_EXPERT_SYS.LISP#> into package SWIM
#FS::LISPM-PATHNAME "LEWIS: BENAVIDES.SWIM; SWDB_EXPERT_SYS.LISP#16"
> (swim event-queue part-list model-list ks-def-list swim-frame)
*****
*                                     *
*                                     *
*          S.W.I.M.                  *
*                                     *
*          -or-                      *
*                                     *
*      Simulate What I Mean         *
*                                     *
*                                     *
*      Designed by John Benavides    *
*****
Run in demo mode? (Y or N) Yes.
Stop on every activation of a Knowledge Source? (Y or N) Yes.
Trace the Knowledge Source call? (Y or N) Yes.
=====SEARCHING-FOR-INSTANCES-OF-KS-DEFINITION=====
KS_Def{MUX-KS}
"Domain definition:"
(TREE
({(OR4){({AND3} IN1 ({INV} SEL0) ({INV} SEL1))
({AND3} IN2 SEL0 ({INV} SEL1))
({AND3} IN3 ({INV} SEL0 SEL1) ({AND3} IN4 SEL0 SEL1)))
====SUB-CIRCUIT-TRY====
({(OR4)
({AND3}{({INPAD} Net{A,IN,X}) ({INV} ({INPAD} Net{S0,IN,X}))
({INV} ({INPAD} Net{S1,IN,X}))
({AND3}{({INPAD} Net{B,IN,X}) ({INPAD} Net{S0,IN,X})
({INV} ({INPAD} Net{S1,IN,X}))
({AND3}{({INPAD} Net{C,IN,X}) ({INV} ({INPAD} Net{S0,IN,X}))
({INPAD} Net{S1,IN,X})
({AND3}{({INPAD} Net{D,IN,X}) ({INPAD} Net{S0,IN,X})
({INPAD} Net{S1,IN,X}))
---RESULTING-BINDING---
(M-OR Part{CMP78,OR4})(M-AN0 Part{CMP81,AND3})(IN1 Net{AA,CORE,X})
(M-NT0 Part{CMP79,INV})(SEL0 Net{SS0,CORE,X})(M-NT1 Part{CMP80,INV})
(SEL1 Net{SS1,CORE,X})(M-AN1 Part{CMP77,AND3})(IN2 Net{BB,CORE,X})
(M-NT1 Part{CMP80,INV})(M-AN2 Part{CMP76,AND3})(IN3 Net{CC,CORE,X})
(M-NT0 Part{CMP79,INV})(M-AN3 Part{CMP75,AND3})(IN4 Net{DD,CORE,X})
=====
====SUB-CIRCUIT-TRY====
({(OR4)
({AND3}{({INPAD} Net{PADOFF,IN,X}) ({INPAD} Net{S0,IN,X})
({INPAD} Net{S1,IN,X})
({AND3}{({INPAD} Net{B,IN,X}) ({INV} ({INPAD} Net{S0,IN,X}))
({INPAD} Net{S1,IN,X})
({AND3}{({INPAD} Net{C,IN,X}) ({INPAD} Net{S0,IN,X})
({INV} ({INPAD} Net{S1,IN,X}))
({AND3}{({INPAD} Net{D,IN,X}) ({INV} ({INPAD} Net{S0,IN,X}))
({INV} ({INPAD} Net{S1,IN,X}))
```

Figure 29. Initialization trace (first part)

```

-----RESULTING-BINDING-----
NIL
=====
-----RESULTING-INSTANCE-----
KS_Inst{ MUX-KS-1,KS_Def{ MUX-KS } }
"Resulting-Bindings:"
((M-OR Part{ CMP78,OR4 } ) (M-AN0 Part{ CMP81,AND3 } ) (IN1 Net{ AA,CORE,X } )
(M-NT0 Part{ CMP79,INV } ) (SEL0 Net{ SS0,CORE,X } ) (M-NT1 Part{ CMP80,INV } )
(SEL1 Net{ SS1,CORE,X } ) (M-AN1 Part{ CMP77,AND3 } ) (IN2 Net{ BB,CORE,X } )
(M-NT1 Part{ CMP80,INV } ) (M-AN2 Part{ CMP76,AND3 } ) (IN3 Net{ CC,CORE,X } )
(M-NT0 Part{ CMP79,INV } ) (M-AN3 Part{ CMP75,AND3 } ) (IN4 Net{ DD,CORE,X } ))
=====SEARCHING-FOR-INSTANCES-OF-KS-DEFINITION=====
KS_Def{ RAMCNT-REFRESH }
"Domain definition:"
(SET ( { RAMCNT } ))
-----RESULTING-INSTANCE-----
KS_Inst{ RAMCNT-REFRESH-SET,KS_Def{ RAMCNT-REFRESH } }
"Resulting-Bindings:"
((RAM_CONTROL Part{ CMP16,RAMCNT } ))
=====SEARCHING-FOR-INSTANCES-OF-KS-DEFINITION=====
KS_Def{ PAD-SWITCH }
"Domain definition:"
(SET ( { OUTPAD } ) ( { OUTPAD } ) ( { OUTPAD } ) ( { OUTPAD } )
( { OUTPAD } ) ( { OUTPAD } ) ( { OUTPAD } ))
-----RESULTING-INSTANCE-----
KS_Inst{ PAD-SWITCH-SET,KS_Def{ PAD-SWITCH } }
"Resulting-Bindings:"
((PAD1 Part{ CMP65,OUTPAD } ) (PAD2 Part{ CMP64,OUTPAD } )
(PAD3 Part{ CMP53,OUTPAD } ) (PAD4 Part{ CMP54,OUTPAD } )
(PAD5 Part{ CMP61,OUTPAD } ) (PAD6 Part{ CMP62,OUTPAD } )
(PAD7 Part{ CMP66,OUTPAD } ))
==TRANSLATING-IGNITION-STATEMENT-FOR-KS-INSTANCE==
KS_Inst{ PAD-SWITCH-SET,KS_Def{ PAD-SWITCH } }
"Ignition Statement:"
(OR (NETCHANGE Net{ FMUXY,CORE,X } '? '1)
(NETCHANGE Net{ NFMUXY,CORE,X } '? '1)
(NETCHANGE Net{ DDOUT1,CORE,X } '? '1)
(NETCHANGE Net{ DDOUT0,CORE,X } '? '1)
(NETCHANGE Net{ MUXY,CORE,X } '? '1)
(NETCHANGE Net{ MUXY,CORE,X } '? '1)
(NETCHANGE Net{ MUXY,CORE,X } '? '1))
==--INSERTING-KS-PINS-INTO-WDB==--
Net{ FMUXY,CORE,X } Net{ NFMUXY,CORE,X }
Net{ DDOUT1,CORE,X } Net{ DDOUT0,CORE,X }
Net{ MUXY,CORE,X } Net{ MUXY,CORE,X }
Net{ MUXY,CORE,X }
==TRANSLATING-IGNITION-STATEMENT-FOR-KS-INSTANCE==
KS_Inst{ RAMCNT-REFRESH-SET,KS_Def{ RAMCNT-REFRESH } }
"Ignition Statement:"
(NETCHANGE Net{ RREF,CORE,X } '? '1)
==--INSERTING-KS-PINS-INTO-WDB==--
Net{ RREF,CORE,X }
==TRANSLATING-IGNITION-STATEMENT-FOR-KS-INSTANCE==
KS_Inst{ MUX-KS-1,KS_Def{ MUX-KS } }
"Ignition Statement:"
(OR (EQUAL (NET-VALUE Net{ SS0,CORE,X } ) 'X)
(EQUAL (NET-VALUE Net{ SS1,CORE,X } ) 'X))
==--INSERTING-KS-PINS-INTO-WDB==--
Net{ SS0,CORE,X } Net{ SS1,CORE,X }

```

Figure 29. (continue) Initialization trace

Normal Evaluation Trace

The simulation trace in Figure 30 illustrates the simulator performing normal model evaluation. The top section of the figure shows the state of the event-queue. Below the event-queue, the first slice of transitions (for time 10) is processed. A list of parts needing evaluation, including the events stimulating each part, is constructed. After the *part evaluation list* is constructed, a list of KS's needing an inference session is constructed. The trace in Figure 30 shows that no KS triggers are activated; this implies that none of the KS's were affected by the events of this time slice.

After all the parts have been evaluated, a list of resulting events is passed on to the event-queue manager. As explained in Chapter 2, the resulting events will have relative times which will be translated into absolute times when stored in the event-queue.

As an example of a part evaluation, consider the first part in the *part evaluation list*, CMP50. CMP50 is an input pad. An input pad behaves like a buffer; it passes its input logic value to its output after a fixed delay. CMP50's input signal, *REF*, transitions to 0. As a result of CMP50's evaluation, the signal on CMP50's output, *RREF*, is scheduled to go to 0 at eight time units later. The resulting event on signal *RREF* is shown as Event{*RREF*,8,0} under the *New Transitions and Deletions* heading in Figure 30. Similar to CMP50's evaluation, all the other parts in the *part evaluation list* are processed.

```

=====Event Queue
10  Event{A,0,1} Event{B,0,0} Event{C,0,1} Event{D,0,0}
    Event{S0,0,1} Event{S1,0,1} Event{ADD0,0,1} Event{PADOFF,0,0}
    Event{ADD1,0,1} Event{DIN0,0,1} Event{DIN1,0,1} Event{RW,0,0}
    Event{REF,0,0}
100 Event{REF,0,1}
110 Event{REF,0,0}
200 Event{RW,0,1}
300 Event{D,0,1} Event{PADOFF,0,1}
400 Event{A,0,1} Event{B,0,1} Event{C,0,1} Event{D,0,1}
    Event{S0,0,1} Event{S1,0,1}
450 Event{S0,0,X} Event{S1,0,X}
500 Event{S0,0,1} Event{S1,0,1}
550 Event{S0,0,X}
600 Event{S0,0,1}
650 Event{S1,0,X}
700 Event{S1,0,1}

=====Current Transitions
10  Event{A,0,1} Event{B,0,0} Event{C,0,1} Event{D,0,0}
    Event{S0,0,1} Event{S1,0,1} Event{ADD0,0,1} Event{PADOFF,0,0}
    Event{ADD1,0,1} Event{DIN0,0,1} Event{DIN1,0,1} Event{RW,0,0}
    Event{REF,0,0}
"Parts needing evaluation (and their events) "
((Part{CMP50,INPAD} (Event{REF,0,0}))) (Part{CMP49,INPAD} (Event{RW,0,0})))
(Part{CMP48,INPAD} (Event{DIN1,0,1}))) (Part{CMP47,INPAD} (Event{DIN0,0,1})))
(Part{CMP45,INPAD} (Event{ADD1,0,1})))
(Part{CMP87,INPAD} (Event{PADOFF,0,0})))
(Part{CMP46,INPAD} (Event{ADD0,0,1}))) (Part{CMP67,INPAD} (Event{S1,0,1})))
(Part{CMP72,INPAD} (Event{S0,0,1}))) (Part{CMP69,INPAD} (Event{D,0,0})))
(Part{CMP70,INPAD} (Event{C,0,1}))) (Part{CMP71,INPAD} (Event{B,0,0})))
(Part{CMP68,INPAD} (Event{A,0,1})))
"List of KS's which will be started: "
NIL

=====New Transitions and Deletions
Add:  Event{AA,8,1} Event{BB,8,0} Event{CC,8,1} Event{DD,8,0}
      Event{SS0,8,1} Event{SS1,8,1} Event{AADD0,8,1} Event{POFF,8,0}
      Event{AADD1,8,1} Event{DDIN0,8,1} Event{DDIN1,8,1} Event{RRW,8,0}
      Event{RREF,8,0}
Delete: -NONE-

```

Figure 30. Normal evaluation trace

DRAM Refresh KS Trace

The simulation trace in Figure 31a illustrates the DRAM refresh KS in operation. CMP16 is an instance of the RAMCNT primitive that will control the refresh operation. When the signal on CMP16's sixth pin transitions to 1, the KS's trigger

is evaluated and returns true. The true trigger starts an inference session for this KS. The lines which start with the string "(RULE START ...)" indicate that the rule named *START* fired (all the antecedents of the rule were true) and the action which resulted. The *START* rule determines that a DRAM refresh operation has started. The actions of the *START* rule are to kill pending events generated for this KS and to start a fresh self-referencing event to check on the DRAM in the future. The *ERROR* shown in this trace is a misnomer and can be ignored because this is the first time the KS is called and no self-referencing events exist. This message should probably be a warning.

```

=====Current Transitions
108 Event{RREF,108,1}

"Parts needing evaluation (and their events) "
((Part{CMP16,RAMCNT} (Event{RREF,108,1})))

"KS ignition being checked: "
(NETCHANGE Net{RREF,CORE,1}'? '1)

"List of KS's which will be started: "
(KS_Inst{RAMCNT-REFRESH-SET,KS_Def{RAMCNT-REFRESH}})

(RULE START SAYS FORGET %FORGET% (INITIAL-FACT))
(ERROR in function zap-events; Could not find event on: " :ME)
(RULE START USES LISP EXPRESSION (ZAP-EVENTS :ME) AND RETURNS NIL)
(RULE START USES LISP EXPRESSION (CREATE-EVENT :ME :TIME 200) AND
RETURNS Event{KS-Global-Net,200,X})
(RULE START SAYS DONE)

=====New Transitions and Deletions
Add:   Event{RAM{Part{CMP16,RAMCNT}},10,0} Event{KS-Global-Net,200,X}
Delete: -NONE-
```

Figure 31. (a) DRAM refresh KS trace (initially)

```

=====Current Transitions
308 Event{POFF,308,1} Event{DD,308,1} Event{KS-Global-Net,308,X}

"Parts needing evaluation (and their events) "
((Part{CMP83,AND3} (Event{DD,308,1}))) (Part{CMP75,AND3} (Event{DD,308,1})))
(Part{CMP86,AND3} (Event{POFF,308,1})))

"List of KS's which will be started: "
(KS_Inst{RAMCNT-REFRESH-SET,KS_Def{RAMCNT-REFRESH}})

(RULE RAM-NOT-REFRESHED USES LISP EXPRESSION
(CREATE-EVENT (PART-PIN Part{CMP16,RAMCNT} 11) :TIME 1 :VALUE 'X) AND
RETURNS Event{CELLRW,1,X})
(RULE RAM-NOT-REFRESHED SAYS FORGET %FORGET% (INITIAL-FACT))
(RULE RAM-NOT-REFRESHED SAYS DONE)

=====New Transitions and Deletions
Add: Event{FA1,5,1} Event{AN4,5,1} Event{FA4,5,0} Event{CELLRW,1,X}
Delete: -NONE-

```

Figure 31. (b) DRAM refresh KS trace (subsequently)

Two events result from circuit evaluation during this simulator time slice. The first event is created by normal evaluation of the RAMCNT model and the second is created by the DRAM refresh KS. The second event is a self-referencing KS event that will mature 200 ns in the future. When mature, the event will cause another inference session for this KS. The purpose of this second session will be to check on the DRAM's status. If the DRAM is not refreshed within this 200 ns period, the KS will cause the DRAM's values to be set to an X (unknown). If the DRAM does get refreshed before the second event matures, the second event will be killed.

The trace in Figure 31b shows the self-referencing KS event created in Figure 31a reaching maturity. The rule named *RAM-NOT-REFRESHED* fires and an event to change signal *CELLRW*'s value is created. This event will cause the values of the bits of the DRAM to go unknown.

Switching-pads KS Trace

The simulation trace in Figure 32 illustrates the switching-pads KS in operation. Four of the seven output pads (CMP66, CMP65, CMP62, and CMP61) transition to 1 at time 318. This causes an inference session for this KS. As was shown in Figure 29, CMP66, CMP65, CMP62, and CMP61 correspond to the switching-pads KS names PAD7, PAD1, PAD6, and PAD5, respectively. As shown in the rule trace of Figure 32, four of seven output pads are found to be switching simultaneously by the KS.

Because four or more pads switching simultaneously indicates a power distribution problem, the KS creates two compensating events for each of the four switching pads. One event models a period of noise with an unknown value on the output signal of the switching pads. The second event models a return to stability for the output pad. These eight events are shown at the bottom of Figure 32 along with another event generated by normal gate evaluation.

Multiplexer KS Trace

The simulation trace in Figure 33 illustrates the multiplexer KS in operation. Both the select signals, *SS0* and *SS1*, for the top multiplexer of Figure 15a transition to X. These signal values ignite the KS which starts rule processing. In the rule trace of Figure 33, two rules fire. The *START* rule retrieves the value of six signals in the KS domain. These are the signals *SS0*, *SS1*, *AA*, *BB*, *CC*, and *DD*. The second rule to fire is called *BOTH-UNKNOWN*. It is true when both select lines are X. This rule's first action is to create an event to resolve the value of the multiplexer subcircuit's output, signal *MUXY*. Subsequent actions stop events generated as a result of both select signals going to X.


```

=====Current Transitions
318 Event{ FMUXY,318,1} Event{ MUXY,318,1}
"Parts needing evaluation (and their events) "
((Part{ CMP66,OUTPAD} (Event{ MUXY,318,1})))
(Part{ CMP62,OUTPAD} (Event{ MUXY,318,1})))
(Part{ CMP61,OUTPAD} (Event{ MUXY,318,1})))
(Part{ CMP65,OUTPAD} (Event{ FMUXY,318,1})))
(Part{ CMP63,INV} (Event{ FMUXY,318,1})))
"KS ignition being checked: "
(OR (NETCHANGE Net{ FMUXY,CORE,1} '? '1)
(NETCHANGE Net{ NFMUXY,CORE,1} '? '1)
(NETCHANGE Net{ DDOUT1,CORE,X} '? '1)
(NETCHANGE Net{ DDOUT0,CORE,X} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1))
"KS ignition being checked: "
(OR (NETCHANGE Net{ FMUXY,CORE,1} '? '1)
(NETCHANGE Net{ NFMUXY,CORE,1} '? '1)
(NETCHANGE Net{ DDOUT1,CORE,X} '? '1)
(NETCHANGE Net{ DDOUT0,CORE,X} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1))
"KS ignition being checked: "
(OR (NETCHANGE Net{ FMUXY,CORE,1} '? '1)
(NETCHANGE Net{ NFMUXY,CORE,1} '? '1)
(NETCHANGE Net{ DDOUT1,CORE,X} '? '1)
(NETCHANGE Net{ DDOUT0,CORE,X} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1))
"KS ignition being checked: "
(OR (NETCHANGE Net{ FMUXY,CORE,1} '? '1)
(NETCHANGE Net{ NFMUXY,CORE,1} '? '1)
(NETCHANGE Net{ DDOUT1,CORE,X} '? '1)
(NETCHANGE Net{ DDOUT0,CORE,X} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1)
(NETCHANGE Net{ MUXY,CORE,1} '? '1))
"List of KS's which will be started: "
(KS_Inst{ PAD-SWITCH-SET,KS_Def{ PAD-SWITCH} })

(RULE START USES LISP EXPRESSION (SETQ ?COUNT 0) AND RETURNS 0)
(RULE START SAYS COUNTED)
(RULE START USES LISP EXPRESSION
(DOLIST (?PAD
(LIST Part{ CMP65,OUTPAD} Part{ CMP64,OUTPAD} Part{ CMP53,OUTPAD}
Part{ CMP54,OUTPAD} Part{ CMP61,OUTPAD} Part{ CMP62,OUTPAD}
Part{ CMP66,OUTPAD})))
(COND
((WENT 1 (PART-PIN ?PAD 1) :WITHIN 5 'NS) (SETQ ?COUNT (+ ?COUNT 1))
(REMEMBER '(?PAD SWITCHING TO HIGH))))))
AND RETURNS NIL)
(RULE START SAYS FORGET %FORGET% (INITIAL-FACT))
(RULE DETERMINE-DELAYS USES LISP EXPRESSION
(SETQ ?DELAY-FACTOR (+ 1 (* (- ?COUNT 2) 0.1))) AND RETURNS 1.2)
(RULE DETERMINE-DELAYS SAYS MODIFY EVENTS)
(RULE DETERMINE-DELAYS USES LISP EXPRESSION
(SETQ ?DELAY-FACTOR (+ 1 (* (- ?COUNT 2) 0.1))) AND RETURNS 1.2)

```

Figure 32. Switching-pads KS trace (first part)

```

(RULE CHECK-PAD1-SWITCHING USES LISP EXPRESSION
 (CREATE-EVENT (PART-PIN Part { CMP65,OUTPAD } 2) :TIME
  (GET-EVENT-FIELD (PART-PIN Part { CMP65,OUTPAD } 2) :TIME) :VALUE 'X)
 AND RETURNS Event { FMUX1,8,X })
(RULE CHECK-PAD1-SWITCHING USES LISP EXPRESSION
 (MODIFY-EVENT (PART-PIN Part { CMP65,OUTPAD } 2) :TIME
  (* ?DELAY-FACTOR
   (GET-EVENT-FIELD (PART-PIN Part { CMP65,OUTPAD } 2) :TIME)))
 AND RETURNS Event { FMUX1,9,6,1 })
(SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON Part { CMP65,OUTPAD })
(RULE CHECK-PAD1-SWITCHING USES LISP EXPRESSION
 (PRINT '(SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON
  Part { CMP65,OUTPAD })) AND RETURNS
 (SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON Part { CMP65,OUTPAD }))
(RULE CHECK-PAD5-SWITCHING USES LISP EXPRESSION
 (CREATE-EVENT (PART-PIN Part { CMP61,OUTPAD } 2) :TIME
  (GET-EVENT-FIELD (PART-PIN Part { CMP61,OUTPAD } 2) :TIME) :VALUE 'X)
 AND RETURNS Event { MUX2,8,X })
(RULE CHECK-PAD5-SWITCHING USES LISP EXPRESSION
 (MODIFY-EVENT (PART-PIN Part { CMP61,OUTPAD } 2) :TIME
  (* ?DELAY-FACTOR
   (GET-EVENT-FIELD (PART-PIN Part { CMP61,OUTPAD } 2) :TIME)))
 AND RETURNS Event { MUX2,9,6,1 })
(SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON Part { CMP61,OUTPAD })
(RULE CHECK-PAD5-SWITCHING USES LISP EXPRESSION
 (PRINT '(SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON
  Part { CMP61,OUTPAD })) AND RETURNS
 (SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON Part { CMP61,OUTPAD }))
(RULE CHECK-PAD6-SWITCHING USES LISP EXPRESSION
 (CREATE-EVENT (PART-PIN Part { CMP62,OUTPAD } 2) :TIME
  (GET-EVENT-FIELD (PART-PIN Part { CMP62,OUTPAD } 2) :TIME) :VALUE 'X)
 AND RETURNS Event { MUX1,8,X })
(RULE CHECK-PAD6-SWITCHING USES LISP EXPRESSION
 (MODIFY-EVENT (PART-PIN Part { CMP62,OUTPAD } 2) :TIME
  (* ?DELAY-FACTOR
   (GET-EVENT-FIELD (PART-PIN Part { CMP62,OUTPAD } 2) :TIME)))
 AND RETURNS Event { MUX1,9,6,1 })
(SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON Part { CMP62,OUTPAD })
(RULE CHECK-PAD6-SWITCHING USES LISP EXPRESSION
 (PRINT '(SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON
  Part { CMP62,OUTPAD })) AND RETURNS
 (SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON Part { CMP62,OUTPAD }))
(RULE CHECK-PAD7-SWITCHING USES LISP EXPRESSION
 (CREATE-EVENT (PART-PIN Part { CMP66,OUTPAD } 2) :TIME
  (GET-EVENT-FIELD (PART-PIN Part { CMP66,OUTPAD } 2) :TIME) :VALUE 'X)
 AND RETURNS Event { MUX0,8,X })
(RULE CHECK-PAD7-SWITCHING USES LISP EXPRESSION
 (MODIFY-EVENT (PART-PIN Part { CMP66,OUTPAD } 2) :TIME
  (* ?DELAY-FACTOR
   (GET-EVENT-FIELD (PART-PIN Part { CMP66,OUTPAD } 2) :TIME)))
 AND RETURNS Event { MUX0,9,6,1 })
(SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON Part { CMP66,OUTPAD })
(RULE CHECK-PAD7-SWITCHING USES LISP EXPRESSION
 (PRINT '(SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON
  Part { CMP66,OUTPAD })) AND RETURNS
 (SIMULTANEOUS SWITCHING BEHAVIOR FOUND ON Part { CMP66,OUTPAD }))
=====New Transitions and Deletions
Add:  Event { NFMUXY,3,0 } Event { FMUX1,9,6,1 } Event { MUX2,9,6,1 }
      Event { MUX1,9,6,1 } Event { MUX0,9,6,1 } Event { FMUX1,8,X }
      Event { MUX2,8,X } Event { MUX1,8,X } Event { MUX0,8,X }
Delete: -NONE-

```

Figure 32. (continue) Switching-pads KS trace

```

=====Event Queue
458 Event{ SS1,458,X } Event{ SS0,458,X }
500 Event{ S0,0,1 } Event{ S1,0,1 }
550 Event{ S0,0,X }
600 Event{ S0,0,1 }
650 Event{ S1,0,X }
700 Event{ S1,0,1 }

=====Current Transitions
458 Event{ SS1,458,X } Event{ SS0,458,X }
"Parts needing evaluation (and their events) "
((Part{ CMP77,AND3 } (Event{ SS0,458,X }))) (Part{ CMP79,INV } (Event{ SS0,458,X })))
(Part{ CMP84,AND3 } (Event{ SS0,458,X }))) (Part{ CMP74,INV } (Event{ SS0,458,X })))
(Part{ CMP85,AND3 } (Event{ SS1,458,X })))
(Part{ CMP86,AND3 } (Event{ SS0,458,X } Event{ SS1,458,X })))
(Part{ CMP73,INV } (Event{ SS1,458,X }))) (Part{ CMP76,AND3 } (Event{ SS1,458,X })))
(Part{ CMP75,AND3 } (Event{ SS0,458,X } Event{ SS1,458,X })))
(Part{ CMP80,INV } (Event{ SS1,458,X })))
"KS ignition being checked: "
(OR (EQUAL (NET-VALUE Net{ SS0,CORE,X } ) 'X)
(EQUAL (NET-VALUE Net{ SS1,CORE,X } ) 'X))
"KS ignition being checked: "
(OR (EQUAL (NET-VALUE Net{ SS0,CORE,X } ) 'X)
(EQUAL (NET-VALUE Net{ SS1,CORE,X } ) 'X))
"List of KS's which will be started: "
(KS_Inst{ MUX-KS-1,KS_Def{ MUX-KS } })

(RULE START SAYS Net{ SS0,CORE,X } IS X)
(RULE START SAYS Net{ SS1,CORE,X } IS X)
(RULE START SAYS Net{ AA,CORE,1 } IS 1)
(RULE START SAYS Net{ BB,CORE,1 } IS 1)
(RULE START SAYS Net{ CC,CORE,1 } IS 1)
(RULE START SAYS Net{ DD,CORE,1 } IS 1)
(RULE START SAYS FORGET %FORGET% (INITIAL-FACT))
(RULE START SAYS NOT DONE)
(RULE BOTH-UNKNOWN USES LISP EXPRESSION
(CREATE-EVENT (PART-PIN Part{ CMP78,OR4 } 5) :TIME 10 :VALUE '1)
AND RETURNS Event{ MUXY,10,X })
(RULE BOTH-UNKNOWN USES LISP EXPRESSION (ZAP-EVENTS
(PART-PIN Part{ CMP78,OR4 } 2))
AND RETURNS T)
(RULE BOTH-UNKNOWN USES LISP EXPRESSION (ZAP-EVENTS
(PART-PIN Part{ CMP78,OR4 } 3))
AND RETURNS T)
(RULE BOTH-UNKNOWN USES LISP EXPRESSION (ZAP-EVENTS
(PART-PIN Part{ CMP78,OR4 } 4))
AND RETURNS T)
(RULE BOTH-UNKNOWN USES LISP EXPRESSION (ZAP-EVENTS
(PART-PIN Part{ CMP79,INV } 2))
AND RETURNS T)
(RULE BOTH-UNKNOWN USES LISP EXPRESSION (ZAP-EVENTS
(PART-PIN Part{ CMP80,INV } 2))
AND RETURNS T)
(RULE BOTH-UNKNOWN SAYS FORGET %FORGET% (NOT DONE))
=====New Transitions and Deletions
Add: Event{ NNS1,3,X } Event{ FA1,5,X } Event{ FA2,5,0 } Event{ NNS0,3,X }
Event{ FA3,5,0 } Event{ MUXY,10,1 }
Delete: -NONE-

```

Figure 33. Multiplexer KS trace

Summary

In this chapter, simulation of an example circuit was illustrated and explained. The example circuit contained simple logic components and more complex memory components. Three knowledge sources were used in the example simulation. The first KS was an expert on simultaneously switching output pads and was used to adjust simulation results if multiple pads switched simultaneously to emulate power supply problems. The second KS was an expert on reconvergent signals in a four input multiplexer and was used to increase simulation accuracy by considering inter-circuit signal relationships to evaluate the behavior of a multiplexer circuit. The third KS was an expert on DRAM refresh periods and was used to change a DRAM's contents to unknowns when not refreshed correctly. Traditional gate-level simulators only use model analysis to simulate a circuit. SWIM used both model analysis and knowledge sources to steer the traditional components of the simulator into performing with increased accuracy.

CHAPTER IV

CONCLUSION

This thesis has described a knowledge-based simulator, SWIM, which combines expert system and digital-circuit simulator technologies. Using this simulator, a circuit designer can simulate circuit behavior phenomena which differs from the standard digital circuit behavior. These phenomena are not simulatable with standard digital circuit simulators and the conditions leading to the phenomena are only recognized by expert designers. The expert system part of the simulator recognizes conditions leading to phenomenal behavior, determines their consequences, and manipulates the simulator in order to emulate the phenomenal behavior.

The knowledge source characterization of knowledge is attractive because it collects all information for one behavior into a single package. This lends itself to maintainability because rules are not dispersed throughout a large unwieldy rule base. Also, the knowledge source package facilitates reusability because knowledge sources can be grouped in libraries defined for specific circuit technologies where they are applicable. As a library grows, so does the power of the simulator.

Knowledge sources differ from the objects in an object-oriented system in the way knowledge is applied. Knowledge sources apply knowledge bottom-up, using pattern recognition. Objects apply knowledge top-down, using inheritance. A useful enhancement to SWIM would be to modify parts to be objects. The object-oriented paradigm lends itself to reusable code and enhances understandability.

Currently, only slightly hard phenomenal behavior has been simulated with the SWIM prototype. Knowledge for the knowledge sources constructed was taken from design guides and textbooks aimed at circuit design. Translating this knowledge into rules and verifying its performance has been the most difficult facet of this project.

Two limitations exist with the current TREE domain definitions described in Chapter 3. First, the assumption that parts in the pattern have only one output. Second, TREE currently can not handle cycles in the definition. Future work is needed to define a method to describe a circuit pattern for parts with multiple outputs and detecting cycles in the definition.

In the course of implementing the SWIM prototype several changes were made to the initial architecture. The initial architecture assumed a strict adherence to the blackboard formalism. The SWIM prototype actually used a modified blackboard architecture in which an evaluator and a world data base can be identified as independent components of SWIM and not part of the blackboard. This change made it possible to couple the traditional parts of a simulator to the expert system. In addition, SWIM did not use the blackboard for all data and control interaction between the knowledge sources and the other SWIM components as initially planned. This change permitted the knowledge sources greater access to circuit information and increased rule flexibility. Finally, SWIM permitted the use of LISP function calls in its rule language; this violated the rule-based formalism, but eased the construction of a rule language.

Future work on the SWIM prototype should center on behavioral knowledge verification, developing an interface for knowledge acquisition/manipulation, physical translation tools, and increasing the level of component model abstraction. The behavioral knowledge is the most important part of this knowledge-based system

because it directly influences its performance. It is impractical to manufacture a circuit to verify every iota of behavioral knowledge; therefore, a more feasible alternative must be investigated.

The current knowledge acquisition/manipulation interface of SWIM is basically the ZMACS editor on the TI Explorer. All knowledge must be coded manually. A user friendly interface which takes advantage of currently existing human factors research should be investigated. An interface which is flexible and not too complicated would be ideal. After several sessions with the interface, a user should not become hindered by it. A graphic interface would probably be preferred for specifying a knowledge source's domain description.

Presently, there is no automatic way to translate a circuit from its structural view to a physical representation. Entering this information by hand prohibits simulating larger circuits. Acquiring access to place and route software for creating a physical representation of a circuit is a solution which should be pursued. Place and route software facilitates the building of a floor-plan for laying out circuit components and attaching wires between them to realize the physical circuit.

Finally, increasing the level of model abstraction to a functional or behavioral level should also be considered because circuit design is evolving toward these more abstract levels. IC sizes are continuing to shrink and entire systems can now be built on an ultra-large integrated circuit. In order to handle this ultra-large system integration (ULSI), *circuit* design is now moving toward *system* design methodologies.

The concept of combining an event-driven simulator and knowledge-based expert system demonstrated in SWIM can be applied in other simulation domains. Some simulation domains that share the characteristics of (1) being built from

primitive parts and (2) having their whole system's behavior affected by both formal and informal relationships are:

- o computer network simulations,
- o battle field simulations,
- o factory system simulations,
- o business decision system simulations,
- o organizational simulations, and
- o manufacturing simulations.

In SWIM's domain, digital circuits, systems are constructed from elementary primitives, gates, whose behavior is based on formal and informal relationships. Formal relationships are defined in the world data base and informal relationships are defined in knowledge source rules. For example, in the *switching-pads* knowledge source of Chapter 3, simulated circuit behavior is modified when too many pads switched simultaneously. This is because an informal relationship shared by the pads is their significant dependence on power and the sharing of a common power supply.

The SWIM prototype was successful in demonstrating the ability to use expert knowledge to simulate. In SWIM, the knowledge source approach was used to simulate small real world circuits successfully. This demonstrates the utility of an artificial intelligence based approach to digital circuit simulation.

APPENDIX

LISP MODEL FOR RAMCELL

```

(defun ramcell_behavior (pins-list events)
  "Behavior for a positive-edge triggered RAM CELL which acts
   like a D flip flop (similar to SN7474) when enabled with 'Select'"
  ;; This function will return a list of any resulting events.
  ;; Input pins 1 (D), 2 (Select), and 3 (R/W or CLK)
  ;; Output pin 4 (Dout or Q)
  (let* ((path1to3 5) ;Delays for signal transitions
         (path_thru_internal_buffers 1)
         (in_trans_ram ;Flag to mark RAM is in transition
          (first (part-ram-list (pin-part (first pins-list))))))
         (storage_ram ;Storage for the value of the RAM CELL
          (second (part-ram-list (pin-part (first pins-list))))))
         (dnet (pin-net (first pins-list))) ;pointer to D pin's signal
         (select (pin-net (second pins-list))) ;pointer to Select pin's signal
         (clknet (pin-net (third pins-list))) ;pointer to CLK pin's signal
         (qnet (pin-net (fourth pins-list))) ;pointer to Q pin's signal
         (cond ((netchange select '? 1 events)
                 (list (make-event :net qnet :Transition (ram-value storage_ram)
                                   :time path_thru_internal_buffers)))
               ((netchange select '? 'X events)
                 (setf (ram-value storage_ram) 'X)
                 (list (make-event :net qnet :transition 'X
                                   :time path_thru_internal_buffers)))
               ((equal (net-value select) 1)
                 (cond ((and (netchange clknet 0 1 events) ;Rising clock edge
                             ;Not in transition?
                             (not (equal (ram-value in_trans_ram) 1)))
                       (setf (ram-value in_trans_ram) 1)
                       (setf (ram-value storage_ram) (net-value dnet))
                       (list (make-event :net qnet :Transition (net-value dnet)
                                         :time path1to3)
                             ;Clear transition flag
                             (make-event :ram in_trans_ram
                                         :Transition 0 :time path1to3)))
                       ((and (netchange clknet '? 'X events) ;Unknown clock edge
                             ;Not in transition
                             (not (equal (ram-value in_trans_ram) 1)))
                       (setf (ram-value in_trans_ram) 1)
                       (setf (ram-value storage_ram) 'X)
                       (list (make-event :net qnet :Transition 'X
                                         :time path1to3)
                             ;Clear transition flag
                             (make-event :ram in_trans_ram
                                         :Transition 0 :time path1to3)))
                       ((ramchange in_trans_ram '? 0 events)
                       (setf (ram-value in_trans_ram) 0))))
               ((equal (net-value select) 0)
                 (list (make-event :net qnet :transition 'Z
                                   :time path_thru_internal_buffers))))))

```

Figure 34. LISP model for RAMCELL

REFERENCES

- Adelsberger, Heimo H., Udo W. Pooch, Robert E. Shannon, and Glen N. Williams. 1986. Rule based object oriented simulation systems. In *Proceedings of the Conference on Intelligent Simulation Environments, January 23-25, 1986*. Simulation Series, no. 17:1, edited by Pual A. Luker and Heimo H. Adelsberger. San Diego, California: Society for Computer Simulation, 107-112.
- Benavides, John A. 1989. A Typical SWIM simulation session. Technical Report presented to The Computer Sciences Department at the University of North Texas, Denton Texas. Photocopied.
- Bloom, Michael. 1987. Mixed-mode simulators bridge the gap between analog and digital design. *COMPUTER DESIGN*, 15 January, 51-65.
- Borden, Andrew, Kao Hugh, Wayne Neyman, and Edward Wojciechowski. 1985. OPS5 as an electronic warfare design tool. In *Artificial Intelligence and Simulation*, edited by Willard M. Holmes. San Diego, California: Society for Computer Simulation, 71-75.
- Culbert, Chris. 1987. *CLIPS Reference Manual*. NASA Johnson Space Flight Center. Published by Computer Software Management and Information Center (COSMIC), The University of Georgia, Athens, Georgia.
- Dillinger, Thomas E. 1988. *VLSI Engineering*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Hayes-Roth, Frederick, Donald A. Waterman, and Douglas B. Lenat. 1983. *Building Expert Systems*. Reading, Massachusetts: Addison-Wesley.
- GenRad. 1985. *HILOS Users Manual*. GenRad Inc., Santa Clara, California.
- Kim, Tag Gon and Benard P. Zeigler. 1988. The Class Kernal-Models in DEVS-Scheme: A Hypercube Architecture Example. *SIMULETTER* 19:2 (June), 20-30.
- Klahr, Philip. 1986. Expressibility in ROSS: An object-oriented simulation system. In *Proceedings of the European Conference at the University of Ghent on February 25-28, 1985*. Simulation Series, no. 18:1, edited by E.J.H. Kerckhoffs, G.C. Vansteenkiste, and Benard P. Zeigler. San Diego, California: Society for Computer Simulation, 136-139.

- Klahr, Philip, David McArthur and Sanjai Narain. 1982. SWIRL: An Object-oriented Air Battle Simulator. *Proceedings of the National Conference on Artificial Intelligence (AAAI-82) August 18-20, 1982 at Pittsburgh Pennsylvania*, 331-334.
- Klahr, Philip, and Donald A. Waterman. 1986. *Expert Systems: Techniques, Tools, and Applications*. Reading, Massachusetts: Addison-Wesley.
- Lightner, Michael R. 1987. Modeling and Simulation of VLSI Digital Systems. *Proceedings of the IEEE* 75:6 (June), 786-796.
- McCluskey, Edward J. 1986. *Logic Design Principles*. Englewood Cliffs, New Jersey: Prentice-Hall.
- Mentor Graphics. 1986. *Quicksim Family Reference Manual*. Mentor Graphics, Beaverton, Oregon.
- Miczo, Alexander. 1986. *Digital Logic Testing and Simulation*. New York: John Wiley and Sons, Inc.
- Milne, Bob. 1989. Behavioral Models Refine Mixed-Signal IC Design. *Electronic Design*, 13 April, 69-72.
- Moser, Jorege G. 1986. Integration of artificial intelligence and simulation in a comprehensive decision-support system. *SIMULATION* 47:6 (June), 223-229.
- Murray, Karen J., and Sallie V. Sheppard. 1988. Knowledge-based simulation model specification. *SIMULATION* 50:3 (March), 112-119.
- Nagel, Laurence W. 1975. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. ERL Memo No. ERL-M520, University of California, Berkeley.
- O'Keefe, Robert. 1986. Simulation and expert systems - A taxonomy and some examples. *SIMULATION* 46:1 (January), 10-16.
- Reddy, Y.V. Ramana, Mark S. Fox, Nizwer Husain, and Malcolm McRoberts. 1986. The Knowledge-based Simulation System. *IEEE SOFTWARE* 3:2 (February), 26-37.
- Terman Christopher J. 1987. Simulation Tools for VLSI. Chapter 3 of *VLSI CAD Tools and Applications*. Edited by Wolfgang Fichtner, and Martin Morf. Boston, Massachusetts: Kluwer Academic Publishers.
- Texas Instruments. 1987. *Advanced CMOS Logic Designer's Handbook*. Texas Instruments Inc., Dallas, Texas.
- Trimberger, Stephen M. 1987. *An Introduction to CAD for VLSI*. Boston, Massachusetts: Kluwer Academic Publishers.

- Ulrich, Ernst, and Itsuo Suetsugu. 1986. Techniques for Logic and Fault Simulation. *VLSI Systems Design*, October, 68-81.
- Valid Logic Systems. 1985. *ScaldSystem Reference Manual*. Valid Logic Systems Inc., San Jose, California.
- Winston, Patrick H. and Berthold Klaue Paul Horn. 1984. *LISP*. 2d ed. Reading, Massachusetts: Addison-Wesley.

SELECTED BIBLIOGRAPHY

- Benavides, John A. "Pattern Description Interfaces for Digital Logic Circuits." In *Proceedings of 12th Annual Computer Sciences Conference of the Federation of North Texas Area Universities, April 28, 1988 at North Texas State University, Denton, Texas, 1988, 12-18.*
- Benavides, John A. "Knowledge-based Simulation." In *Proceedings of 12th Annual Computer Sciences Conference of the Federation of North Texas Area Universities, April 28, 1988 at North Texas State University, Denton, Texas, 1988, 76-82.*
- Benavides, John A. and Dana L. Wyatt. "Implementing a Knowledge-based Digital Circuit Simulator." Report presented at Society for Computer Simulation's Southwestern Regional Council Conference in Ft. Worth, Texas, October 20-21, 1988. Photocopied.
- Benavides, John A. and Dana L. Wyatt. "Improving Digital Circuit Simulation: A Knowledge-based Approach." In *Proceedings of the 1988 Winter Simulation Conference, December 12-14, 1988, 364-371.*
- Groundwater, Elizabeth H. "A demonstration of an ocean surveillance information fusion expert system". In *Artificial Intelligence and Simulation*, edited by Willard M. Holmes. San Diego, California: Society for Computer Simulation, 1985, 6-9.
- Halley, Marc A., Thomas Miller, Craig Hougum and William Mosenthal. "SONAR PLEXUS - Enhancing a command and control simulation with reasoning." In *Simulators IV: Proceedings of the SCS Simulators Conference on 6-9 April, 1987 in Orlando, Florida*, Simulation Series, no. 18:4, edited by Bruce T. Fairchild. San Diego, California: Society for Computer Simulation, 1987, 15-18.
- Robertson, Paul. "A Rule Based Expert Simulation Environment." In *Proceedings of the Conference on Intelligent Simulation Environments, January 23-25, 1986*. Simulation Series, no. 17:1, edited by Pual A. Luker and Heimo H. Adelsberger. San Diego, California: Society for Computer Simulation, 1986, 9-15.
- Schildt, Herbert. *Artificial Intelligence Using C*. Berkeley, California: Osborne McGraw-Hill, 1987.
- Shortcliffe, Edward H., and Bruce G. Buchanan. *Rule-Based Expert Systems*. Reading, Massachusetts: Addison-Wesley, 1984.