Software engineering considerations in the design and implementation of a minicomputer-based digital logic circuit timing simulator.

Brenda M. Csencsits

Follow this and additional works at: http://preserve.lehigh.edu/etd

Part of the Electrical and Computer Engineering Commons

Recommended Citation


This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.
SOFTWARE ENGINEERING CONSIDERATIONS IN THE DESIGN
AND IMPLEMENTATION OF A MINICOMPUTER-BASED
DIGITAL LOGIC CIRCUIT TIMING SIMULATOR

by

Brenda M. Csencsits

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Electrical Engineering

Lehigh University
1978
This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

Dec 15, 1978
Date

Professor in Charge

Chairman of Department
ACKNOWLEDGMENTS

I am very grateful to Dr. Mark J. Flomenhoft who encouraged my work in logic circuit simulation and to whom I owe much of my basic understanding of this subject.

I am also grateful to Melinda C. Kohn for initiating me to good software engineering techniques.

I thank Professor Patti Ota for her help and advice in writing this thesis.

I also thank Bell Laboratories for its financial support.
TABLE OF CONTENTS

Table of Figures ........................................ vii
Abstract ................................................. 1

1. Introduction ........................................... 2
   1.1 What is a Timing Simulator – Definition Used in This Thesis 2
   1.2 Why Use a Timing Simulator .......................... 2
   1.3 Overview of TSAR .................................... 2
   1.4 Overview of Thesis ................................. 3

2. Software Engineering Considerations ............... 5
   2.1 Production Environment Usage ...................... 5
   2.2 Minicomputer Layout and Design Aids System .... 5
   2.3 Operating System .................................... 6
   2.4 User Orientation .................................... 7
   2.5 Software Techniques ................................. 7
      2.5.1 Modular Design ................................ 8
      2.5.2 Working Path Approach ......................... 9
      2.5.3 Incremental Bottom-up Implementation ....... 10
      2.5.4 Structured Documentation ...................... 11
      2.5.5 Data Access Routines .......................... 12

3. Capabilities ........................................... 14
   3.1 3-Valued Logic .................................... 14
   3.2 Fault-Free ........................................ 14
   3.3 Flip-Flop Race ..................................... 15

- iv -
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1</td>
<td>A Critical Race Condition for a NAND Flip-Flop</td>
<td>15</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>A NAND Flip-Flop Circuit Named FF</td>
<td>21</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>A Standard Simulation Results File</td>
<td>28</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>A Simulation Results File Using the &quot;SET&quot; Command</td>
<td>29</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>A Simulation Results File Using the &quot;MONITOR&quot; Command</td>
<td>29</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>A Simulation Results File Using the &quot;TRACE 1&quot; Command</td>
<td>30</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>A Simulation Results File Using the &quot;SWITCH&quot; Command</td>
<td>31</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Main Memory Map of the Circuit Description Tables</td>
<td>33</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Topology Word as Stored in Main Memory</td>
<td>34</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Pointers and Inputs/Fanouts Lists as Stored in Main Memory</td>
<td>35</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Current Value as Stored in Main Memory</td>
<td>36</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Propagation Delays as Stored in Main Memory</td>
<td>37</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>Schedule Table Entry as Stored in Main Memory</td>
<td>37</td>
</tr>
<tr>
<td>Figure 6.7</td>
<td>Flip-Flop Pairs as Stored in Main Memory</td>
<td>38</td>
</tr>
<tr>
<td>Figure 6.8</td>
<td>Circuit Outputs and Monitor Points as Stored in Main Memory</td>
<td>39</td>
</tr>
<tr>
<td>Figure 7.1</td>
<td>A NAND Flip-Flop Circuit Named FF</td>
<td>42</td>
</tr>
<tr>
<td>Figure 7.2</td>
<td>Circuit Description Data for Figure 7.1</td>
<td>43</td>
</tr>
<tr>
<td>Figure 7.3</td>
<td>Initial Conditions of the NAND Flip-Flop</td>
<td>44</td>
</tr>
</tbody>
</table>
Figure 7.4  Input Stimuli Are Applied to the NAND Flip-Flop  

Figure 7.5  The Response of the NAND Flip-Flop to the Input Stimuli  

Figure 7.6  Data Structure for the Fixed Increment Time Flow Mechanism  

Figure 8.1  Elapsed Time Required to Simulate Some Sample Logic Circuits  

Page  

44  

45  

46  

50
Abstract

A computer program that simulates the fault-free behavior of logic circuits using a variable delay timing model is implemented on the HP 21M-series minicomputer under the Hewlett-Packard RTE-III operating system.

The circuit is described as an interconnection of logic gates, each of which may be assigned its own propagation delays: its nominal high-to-low delay and its nominal low-to-high delay.

The simulation program reports circuit oscillations, spikes, and critical races in flip-flops.

Considerations of implementation of such a program on a minicomputer are discussed. These include low computer cost, limited memory, slower machine cycle time, efficient data structure, and slower input-output on secondary storage.

Software engineering considerations are discussed. The simulator is a FORTRAN program which is modular, uses data access routines, and is structured by pseudo-language comments.

The work reported here was the author's work assignment carried out at Bell Laboratories, Allentown, Pennsylvania, during 1977-1978.
CHAPTER 1

INTRODUCTION

1.1 What is a Timing Simulator

In this thesis, "timing simulator" is defined to be a computer program that predicts a digital logic circuit's response to a sequence of input stimuli, given the propagation delays for each element of the circuit. The logic circuit is modeled as an interconnection of logic functions. The sequence of input stimuli is a list of logic-0 and logic-1 levels presented through time to the inputs of the logic circuit. The propagation delays model the behavior of the physical function blocks of the circuit.

1.2 Why Use a Timing Simulator

In some technologies, like Integrated Injection Logic (IIL), a circuit's performance is highly dependent on parasitic capacitance. When these capacitances can be lumped together and assigned as a propagation delay through an element, a logic circuit simulator can more accurately predict the performance of the circuit than by simplistically assuming some universal unit of delay.

1.3 Overview of TSAR

TSAR (Timing Simulator to Analyze Response) is a FORTRAN program that simulates the response of a logic circuit to a sequence of input stimuli. The logic circuit is described as an interconnection of basic logic functions (AND, NAND, OR, NOR). An assignable delay timing model is used. Each gate may be assigned a pair of delays: its nominal
high-to-low propagation delay and its nominal low-to-high propagation
delay. TSAR assumes that no faults are present in the circuit, although
the capability to simulate "gate output stuck-at-0(1)" faults exists.
TSAR uses three values of logic: logic-0, logic-1, and indeterminate
(don't-know). Don't-know is used to initialize the circuit and to
resolve a flip-flop race, a circuit oscillation, or a spike.

For each new input stimuli, TSAR allows a user to observe
the state of circuit outputs and internal monitor points when the
circuit settles, as well as during the transition of the circuit from
its initial state to its final state.

TSAR runs on a Hewlett-Packard 21M-series minicomputer with
64K-words of main memory, under the Real Time Executive (RET)-III
operating system.

1.4 Overview of Thesis

The remainder of this thesis is organized as follows:

Chapter 2 considers several aspects in engineering a software
project, like TSAR. The software was required to fit into a production
minicomputer layout and design aids system. The design and development
of the program was affected by the operating system and such software
techniques as modular design, working path approach, incremental
bottom-up implementation, structured documentation, and use of data
access routines.

Chapter 3 details the capabilities of TSAR. The capabilities
include dealing with three values of logic, considering only fault-free
circuits, resolving flip-flop races, circuit oscillations, and spikes,
and monitoring internal points.
Chapter 4 describes the input to TSAR. Input consists of circuit description tables, a sequence of input stimuli, a list of gates and their respective propagation delays, and user commands. Presently, there are seven user commands: MONITOR, QUIT, RUN, RUN 'n', SET, SWITCH, and TRACE 'n'.

Chapter 5 describes the output from TSAR. Output consists of the circuit's response to each input stimulus. The standard way in which TSAR records the circuit response may be altered by using the SET, MONITOR, TRACE, or SWITCH commands.

Chapter 6 describes the data as it is resident in main memory.

Chapter 7 steps through the fixed increment time flow mechanism that drives the simulator through time.

Chapter 8 summarizes the performance of TSAR.

Chapter 9 documents the program limits.

Chapter 10 recognizes improvements that could be made in TSAR such as allowing larger circuits, implementing a clocking capability for sequences of input stimuli, and providing a fault simulation capability.
CHAPTER 2

SOFTWARE ENGINEERING CONSIDERATIONS

This chapter discusses several considerations that were made when TSAR was designed. These considerations included: the production mode in which TSAR would run, the system of design aids of which TSAR would be a part, the operating system under which TSAR would run, the user-orientation required by a computer program to run in production mode, and the software techniques used to develop TSAR.

2.1 Production Environment Usage

TSAR was required to be a tool for integrated circuit designers to use in their routine design of large-scale integrated circuits, from logic and design verification to testing. To meet this requirement, TSAR was engineered to provide fast turnaround and be easy to use.

2.2 Minicomputer Layout and Design Aids System

TSAR was needed to be part of a system of design aids (REGAL) for integrated circuit layout. REGAL runs on an HP 21M-series minicomputer to provide a low-cost system for integrated circuit design. The HP 21M-series minicomputer hardware imposes memory limits, longer machine cycles, and slower secondary I/O not usually associated with large mainframe computers. Therefore, TSAR was optimized to overcome these drawbacks by packing data as much as possible into main memory.

Some generality is lost by depending on support programs provided by REGAL. That is, TSAR itself is not a stand-alone software package.
2.3 Operating System

TSAR was developed and runs under the Hewlett-Packard Real Time Executive (RTE-III) operating system. This section describes how RTE-III affected the development and implementation of TSAR.

RTE-III allocates to a user program a maximum main memory address space of 18K (16-bit) words. An 18K partition is not large given that the circuit description data must be memory resident to optimize the simulator's performance. Therefore, TSAR is divided into a main program and five segments that:

1. Load the circuit description
2. Load the sequence of input stimuli
3. Load the propagation delays
4. Read and execute a user command
5. Apply a new input pattern and simulate the circuit response

The main program and the circuit description data are always memory resident. A segment is called in by the main program as needed. With this scheme, the circuit size is still limited to about 400 gates. This limitation is expected to be overcome when the RTE-IV operating system becomes available. RTE-IV is specified to provide larger (about 26K words) user program partitions and Extended Memory Addressing (EMA). The larger user program partitions naturally will allow larger circuits to be simulated.

---

1Hewlett-Packard-supplied RTE-III manual.
But, even larger circuits can be handled by using EMA. EMA allows a user program to address memory outside its partition. Therefore, the circuit description data can spill over into the extended area of main memory. The drawback is that an access to extended memory is slower than an access to memory within the program's partition. However, an access to extended memory is expected to be faster than a disc access. A degree of portability is lost is using EMA because the HP FORTRAN compiler passes call-by-value rather than (standard) call-by-name parameters when addressing the extended memory area. Call-by-name is a means of passing data to a module by passing the address of the data. Call-by-value is a means of passing data to a module by passing a copy of the data.

2.4 User Orientation

In order for TSAR to be used routinely by engineers, TSAR was required to be highly user-oriented. TSAR is user-oriented because it is easy to learn and use, it is "user-proof", and it provides fast turnaround. The simple command language (as described in Chapter 4) provides the user with a few simple but powerful commands. TSAR protects itself from a user by checking consistency of data and user input.

2.5 Software Techniques

This section describes how some software design concepts were used in the design and implementation of TSAR. The techniques to be discussed are:

---


1. Modular design
2. Working path approach
3. Incremental bottom-up implementation
4. Structured programming
5. Structured documentation
6. Data access routines

2.5.1 Modular Design

The goal in the design of TSAR was to break the problem (simulator) up into manageable, well-defined pieces (modules). Breaking the problem up was an iterative process. Each time through the iteration, modules were refined into less compound functions. This refinement technique is known as successive refinement. In most cases, the modules are well-defined and manageable; they perform a simple function involving a few lines of code. On hindsight, however, some modules could have been broken up into smaller pieces.

The motivation for modularity is maintainability. Comprehension of a module is a major factor in being able to maintain (fix and debug) it. According to studies by Weinberg, comprehension of a module drops quickly if the module is bigger than thirty statements.

However, having small pieces is not enough. The pieces must be as independent (decoupled) as possible. Modules are coupled through

---

3 Kernighan, p. 21.
data. A module is most decoupled from other modules when it knows
only about the data that is necessary for it to accomplish its task.
The data coupling that is present should be highly visible. FORTRAN
subprogram calling parameters and labeled common are two ways to keep
coupled data visible.

2.5.2 Working Path Approach

When the simulator was designed and the modules were identi-
ifed, a "working path" was identified which was to be implemented first.
The working path was the thinnest thread through the system that would
allow a user to simulate the circuit response to a sequence of input
stimuli. The advantages of implementing a thin thread of the systems
are:

1. It proves in the algorithms as early as possible.
2. Major interfaces are exercised early in the project.
3. It allows real circuits to be simulated as early as
   possible.
4. It allows user feedback as early as possible.

The working path chosen for TSAR was:

1. Load the circuit description.
2. Load the sequence of input stimuli.
3. Simulate the circuit's response to the entire
   sequence of input stimuli.

Each of these three tasks in turn was implemented by choosing
a thin thread. The circuit description data was loaded from mag tape
and minimal error-checking was done. The sequence of input stimuli was
loaded from mag tape, again, with minimal checking. Only input stimuli were processed — comments and the command to reset the oscillation limit were enhancements to be added later.

One command (RUN) was implemented to apply the entire sequence of input stimuli to the circuit and print the results on the line printer. When a circuit oscillation, flip-flop race, or spike was detected, the program simply stopped. The modules to resolve these conditions were added later. When the end of the sequence of input stimuli was reached, again, the program simply stopped.

Notice that specifying propagation delays was not implemented in the thin thread implementation. A zero- and unit-delay timing model was used so that results could be checked against the results of an existing zero- and unit-delay simulator.

Consequently, the first version of TSAR performed a crude simulation: the user could only interact with the system to a minimum degree; simulation would not proceed if a circuit oscillation, flip-flop race, or spike was detected; and a zero- and unit-delay timing model was used. The enhancements were added later one-by-one.

2.5.3 Incremental Bottom-Up Implementation

Much of the TSAR system was implemented incrementally from the bottom up. Implementation here means coding, testing, and debugging. Implementing a system incrementally means adding one new module to the system at a time and then testing the system with the new module added.\(^\text{"}\)

\(^\text{*Yourdon, p. 342.}\)
This approach contrasts a phased implementation where several modules are added to the system and then tested for their interaction in the system. Clearly, the incremental approach limits the amount of new information a programmer must deal with when testing and debugging the system.

The system was able to be implemented from the bottom up because the lowest level modules were well-defined. The drivers used to test the modules were often skeleton forms of the actual modules to be implemented at the next higher level.

2.5.4 Structured Documentation

Structured documentation is defined here to mean a systematic way of generating and retrieving documentation for a program or subprogram.\footnote{Melinda C. Kohn, "Structured Documentation Approach", Bell Laboratories Technical Memorandum TM-76-8614-4, 1976.} Documentation should include:

1. Definition of the (sub)program and date written
2. Definition of each calling parameter
3. Definition of each internal variable
4. Definition of each file
5. High-level description of the algorithm using a limited number of constructs for flow of control

One approach to structured documentation is to include the documentation as comments in the (sub)program. Documenting with comments becomes more systematic when a unique key from a predefined set of keys begins the comment. The key denotes what the particular documentation is.
For example, in FORTRAN, a comment line begins with "C". So "CDV" may be the key used to denote the definition of an internal variable. In particular,

\[
\text{CDV ICHAR - ASCII character in Al format (integer)}
\]

\[
\text{C;}
\]
defines the variable ICHAR to be an integer variable that holds an ASCII character. "C;" ends the definition.

It becomes clear that, when this standard for documentation is followed, the documentation is retrievable using a string editor. For example, a search on the string "CDV ICHAR" gets the first line of the definition of the variable ICHAR.

A systematic way of generating the documentation is to create a template as an editable file. This file is then filled in when a new (sub)program is written.

This approach to documentation has two shortcomings. First, the keys and standards for using them must be known and followed by the person maintaining the (sub)program. Second, this approach requires discipline by the programmer and maintainer to adhere to the standards.

2.5.5 Data Access Routines

A data access routine fetches/stores a piece of data from/to the data base. Using data access routines isolates the data structure from the program — only the access routines need to know the physical structure of the data base. There are several advantages to using data access routines:
1. Access to the data base is limited to these routines, thereby decreasing the probability of "clobbering" the data base, i.e., they improve data integrity.

2. The physical data structure of the data base is changed more readily. Only the access routines are rewritten to accommodate the change.

In TSAR, there is an access routine to fetch/store each of the following pieces of data:

1. Fanin number of an element
2. Fanout number of an element
3. Type of an element
4. Propagation delays of an element
5. Name of an element
6. An input stimulus
7. Pointer to an element's fanin/fanout list
8. The fanin/fanout list of an element
CHAPTER 3
CAPABILITIES

TSAR uses three values of logic to simulate the fault-free circuit response to a sequence of input stimuli. TSAR can also recognize and resolve flip-flop races, circuit oscillations and spikes. Further, TSAR allows a user to monitor internal points in the circuit. Each of the underlined topics is discussed in this chapter.

3.1 3-Valued Logic

TSAR is a 3-valued logic simulator: each element in the circuit is recognized to be at a logic-0 level, a logic-1 level, or at an indeterminate (don't-know) level. Internally to the program, two bits are used to represent these values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic-0</td>
<td>00</td>
</tr>
<tr>
<td>Logic-1</td>
<td>11</td>
</tr>
<tr>
<td>Don't-know</td>
<td>01</td>
</tr>
<tr>
<td>(Undefined)</td>
<td>10</td>
</tr>
</tbody>
</table>

In this thesis, the 2-bit notation is shortened to be 0, 1, and 3 (or X), respectively.

3.2 Fault-Free

TSAR simulates the circuit response to a sequence of input stimuli assuming that no faults are present in the circuit. Under this assumption, TSAR is used to verify the performance of a circuit design with respect to response time, race conditions, and spike conditions.
The simulation of faulty circuits in parallel with the good circuit has not been implemented, although the capability to do so exists. The fault model would be that the output signal of a single gate is stuck-at-0 or stuck-at-1. The functions that need to be implemented are fault list generation, fault injection, fault detection, and fault dropping. Fault list generation is the process of determining the possible faults in a circuit assuming some fault model like singly-occurring stuck-at faults. Fault injection is the process of incorporating the fault(s) in the circuit. Fault detection is the process of determining when the faulty behavior of the circuit is observable. Fault dropping is the process of ignoring a fault once it has been detected some specified number of times.

3.3 Flip-Flop Race

A flip-flop (latch) is defined to be a cross-coupled pair of gates. The discussion here is limited to NAND flip-flops. The dual of this discussion applies to NOR flip-flops.

A "critical race" occurs in a NAND flip-flop when both the "set" and "clear" inputs change simultaneously from 0 to 1 and the delays through the gates are equal. When this simultaneous input change occurs, the outputs of both gates oscillate between 1 and 0 indefinitely. Figure 3.1 illustrates this situation.

```
0 + 1 ——> 1 + 0 + 1....
0 + 1 ——> 1 + 0 + 1....
```

A Critical Race Condition for a NAND Flip-Flop

Figure 3.1
It is important to recognize and resolve this situation in a simulator, since such an oscillation degrades the simulator's performance.

A critical race also occurs as follows (X represents the don't-know value):

"...if a NAND latch is in the "set" state with both its input at 1 and a unit-width 0-pulse occurs on the reset input, the effect is the same as the simultaneous multiple-input change situation. ...the simultaneous change of the set and clear inputs from 0 and X to 1 and 1 ... produces the oscillation 1X + X0 + 1X + X0 + ....."

Summarizing, then, a critical race condition exists whenever the state of the flip-flop is one of the following:

00
OX
X0

Whenever one of these states is recognized, the state is overwritten by XX and, hence, the race is resolved.

By duality, a race condition for a NOR flip-flop is recognized when the state of the flip-flop is one of the following:

11
X1
1X

3.4 Circuit Oscillation

To optimize a logic simulator's performance, it is important to recognize and (artificially) terminate a circuit oscillation. Otherwise, the scheduling of circuit changes would go on indefinitely.

---

In TSAR, a circuit oscillation is declared when the input pattern changes and the circuit has not settled after some "reasonable" number of units of time -- 2000 by default. The default value may, of course, be changed by the user before a new input stimulus is applied to the circuit.

Once a circuit oscillation is declared, it must be terminated so that the simulation can proceed. To terminate an oscillation, the scheduled values of (active) gates are persistently overridden by the don't-know value as the simulation commences. The don't-knoths will propagate and, eventually, the circuit will settle.

Notice that this technique preserves as much of the circuit state information as possible since a don't-know value is selectively given to active gates only. Consequently, oscillation loops should be identifiable by the fact that the gates in the loop are in the don't-know state.

The technique to selectively set only the oscillating gates to don't-know is described as follows:

Given that a logic value is represented by 2 bits:

<table>
<thead>
<tr>
<th>Logic Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>X</td>
<td>10</td>
</tr>
</tbody>
</table>

Let the leftmost bit be Y and the rightmost bit be Z. Let the LAST VALUE of an element be its current value and the NEW VALUE be the logic value just computed to be the value to which the element is about to switch. Parallel simulation means that "n" circuits are simulated in
parallel. The eight Y bits of the logic values are stored in the leftmost byte of the (HP 16-bit) word and the eight Z bits are stored in the rightmost byte. So, given an element

"... that just changed value, the OR of the corresponding Y and Z bits of the exclusive-OR of the LAST and NEW VALUEs is a mask whose 1's define the circuits that changed. Modifying the element's value by OR-ing that mask to the Z bits and AND-ing the complement of the mask to Y bits imposes the appropriate X's."  

3.5 Spikes

The function of the spike analysis is to simply determine if the transition of a gate's output signal initiated by the present input change in opposition to the transition initiated by a past input change. TSAR recognizes input conditions on a gate that may cause it to emit an erroneous pulse using the spike analysis technique described by Thompson, et al. Briefly, if inputs to a gate change faster than the gate's response time, a "spike" is declared.

The user may choose to override the response to a short input pulse with don't-know or to ignore the pulse. In the former case, the "new-value" of a gate (which was scheduled in response to the leading edge of the input pulse) is overridden with don't-know. In the latter case, the event is unscheduled. The choice depends on the confidence a user has in the nominal propagation delays that are used.

---

11Flomenhoft, p. 257.


- 18 -
3.6 Observe Internal Points

In addition to observing the circuit outputs, TSAR allows a user to observe points internal to the circuit. This feature allows a user to more easily debug the circuit when race, oscillation, and spike conditions occur. For example, when TSAR reports that a flip-flop race has been resolved, a user may monitor the driving gates of that flip-flop to more readily see how and when the race condition comes about.

A user interactively specifies the monitor points during the simulation session. That is, that part of the source circuit description which lists the outputs does not have to be changed to include additional points and then re-input to TSAR.
CHAPTER 4

INPUT PACKAGE

The input package consists of four items:

1. Circuit description tables which describe the interconnection of the gates in the circuit.
2. A sequence of input stimuli which describes the logic values presented to the circuit inputs.
3. Propagation delays for a group of gates. Each gate may have its own nominal low-to-high and nominal high-to-low propagation delays specified.
4. User commands.

Each of these four items which comprise the input package is described in detail in the following sections.

4.1 Circuit Description Tables

The circuit description tables describe the circuit in terms of gate interconnectivity in a tabular form. These tables may be built by a computer program that processes a "source" circuit description language. This computer program must determine the gate-level interconnection of the circuit and build a file composed of a header and five tables. (Throughout the file, data in parentheses and lines beginning with an asterisk are comments.)

The file header (the first twelve lines of the file) specifies the number of circuit elements (inputs + batteries + gates), the number of primary inputs, the number of primary outputs, the number of pairs
of elementary flip-flops, and the length of each "connectivity list". As an example, consider the logic diagram shown in Figure 4.1.

Circuit:  FF

\[
\begin{array}{c}
S \\
S2 \\
R \\
R2 \\
\end{array}
\begin{array}{c}
Q \\
Q2 \\
\hline
\end{array}
\]

A NAND Flip-Flop Circuit Named FF

Figure 4.1

The circuit description file header is as follows:

\*
* FF
* No. elements
6
* 2 elements have fanin >1
* No. primary inputs
4
* No. primary outputs
2
* No. NDFF & NRFF pairs (No. of elementary flip-flops)
1
*
6

The five tables are as follows:

1. Element characterizations - delay, fanin, fanout, type
2. Flip-flop race pairs
3. Input connectivity
4. Output connectivity
5. Primary outputs

The element characterization table specifies the delay or battery value, fanin, fanout, type, and name of each circuit element. This table is headed by the comment "* DELAY-FANIN-FANOUT-TYPE-NAME", and the data for the i-th element is line \( i+13 \) of the file. The element characterization table for the example is as follows:

- 21 -
* DELAY-FANIN-FANOUT-TYPE-NAME

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(2)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(3)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(4)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(5)</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(6)</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

For each element, the number in parentheses at the beginning of the line gives the element number. The next four data give the propagation delay or battery value (0 or 1), the fanin, the fanout, and the type (0-Input, 1-Battery, 2-AND, 3-NAND, 4-OR, 5-NOR). The last datum is the name (strictly, the output-net-name) of the element. Note that the primary inputs appear as the first elements in the table. Also, connection to a primary output does not count as "fanout". Lastly, gates with zero fanin are present as batteries. Zero-fanin ANDs and NORs are present as one-batteries; zero-fanin ORs and NANDs are present as zero-batteries.

The second circuit description table designates flip-flop race pairs in terms of element numbers, where the i-th data pair defines the i-th flip-flop. The order of the data pairs and the order of element numbers in a pair are not significant. The flip-flop race pairs table for the example is:

* FLIP-FLOP PAIRS

5
6

The third and fourth tables are the input and output connectivity lists for the circuit. In each of these tables a line begins with an element number enclosed in parentheses, followed by the numbers of the inputs (third table) or outputs (fourth table) of the element. The connection lists for high-fanin and high-fanout elements will extend onto
the next line without the element number repeated. Elements with zero fanin (e.g., primary inputs) or zero fanout (e.g., unused gates) are absent from the appropriate table. The connectivity tables for the example are:

* INPUT CONNECTIVITY LIST
(5) 1 2 6
(6) 3 4 5

* OUTPUT CONNECTIVITY LIST
(1) 5
(2) 5
(3) 6
(4) 6
(5) 6
(6) 5

The last table is a list of the numbers of the primary outputs in the order they are to appear in the output values listings. The primary outputs table for the example is:

* PRIMARY OUTPUTS
  5
  6

4.2 Sequence of Input Stimuli

The input sequence file is a list of input stimuli (patterns). Each pattern specifies logic values for all of the circuit inputs at a particular time. Each pattern in the sequence is specified by the letter "T", an apostrophe, the input values, and another apostrophe:

T<input values>'

The input values are represented by 0 for logic-0, 1 for logic-1, and 3 for don't-know. The number of values specified must match the number of circuit inputs. For each pattern, the input values are ordered the same way as listed in table 1 of the simulation tables file.
The FF circuit has four inputs. Table 1 of the simulation tables lists the inputs in the following order:

\[\begin{align*}
S \\
S2 \\
R \\
R2
\end{align*}\]

The order of the list affects how the values specified in a pattern are assigned to the circuit inputs. The input pattern T'1011', therefore, assigns the values in the following way:

\[\begin{align*}
S &= 1 \\
S2 &= 0 \\
R &= 1 \\
R2 &= 1
\end{align*}\]

An example of a sequence of input patterns for the FF circuit is:

\[\begin{align*}
T'0000' \\
T'0001' \\
T'0010' \\
T'0011' \\
T'0100' \\
T'0101' \\
T'0110' \\
T'0111' \\
T'1000' \\
T'1001' \\
T'1010' \\
T'1011' \\
T'1100' \\
T'1101' \\
T'1110' \\
T'1111'
\end{align*}\]

4.3 Propagation Delays

The file of propagation delays is a list of gates identified by name and their respective nominal low-to-high and nominal high-to-low propagation delays. This file is used for circuits whose propagation delays are a function of capacitance, etc. (Integrated injection logic,
for example). The propagation delays file is an optional input to TSAR. If none is specified, TSAR defaults to a zero- and unit-delay timing model.

A gate is assigned a pair of propagation delays by writing

\[ \text{<gate name> <low-to-high> <high-to-low>} \]

where low-to-high is the nominal low-to-high propagation delay and high-to-low is the nominal high-to-low propagation delay. Each delay may range from 0 through 255 units. For example,

\[ \text{QB 78 20} \]

assigns to QB a low-to-high propagation delay of 78 units and a high-to-low propagation delay of 20 units. If a gate is repeated more than one time in the list of delays, the last assignment listed is the one that remains in effect.

### 4.4 User Commands

This section is a summary of the seven commands available in TSAR (in alphabetical order).

The commands may be grouped by function as follows:

1. **Execution**: The "RUN" and "RUN 'n'" commands accomplish a simulation of the fault-free circuit.

2. **Enhance the results report**: The "MONITOR", "SET", "SWITCH", and "TRACE 'n'" commands each modify the manner in which TSAR records the simulation results.

3. **Exit**: The "QUIT" command returns control to the operating system.

Any of the commands may be issued in response to the prompt "?".
The seven commands are:

**MONITOR**  Add to the current list of circuit outputs those elements subsequently typed in by the user. A null response (carriage return) ends the prompt for additional elements.

**QUIT**  Return control to the operating system.

**RUN**  Apply the remaining input patterns to the fault-free circuit.

**RUN 'n'**  Apply the next 'n' input patterns to the fault-free circuit, unless the end of the input sequence is reached first.

**SET**  Define the symbols that represent the logic-0, logic-1, and don't-know states.

**SWITCH**  Record the state of the circuit outputs in the results file only when one or more circuit outputs changes state. SWITCH turns off trace mode.

**TRACE 'n'**  Record the state of the circuit outputs in the results file after every multiple of n units of delay during subsequent simulations. "TRACE 'n'" turns off switch mode. "TRACE 0" turns off trace mode.
The output package is a file of the simulation results. The simulation results file lists the state of the circuit outputs at various times. This chapter describes the "standard" and "non-standard" results file. "Standard" means the way TSAR records the states of the circuit outputs if the "MONITOR", "TRACE", "SET", or "SWITCH" commands have not been used during a TSAR session. "Non-standard" means the way TSAR records the states if any one (or more) of these four commands were issued during a TSAR session.

5.1 The Standard Simulation Results File

The standard simulation results file lists the state of the circuit outputs when the circuit has settled after applying each input pattern. The state of an output is represented by 0 for logic-0, 1 for logic-1, and 3 for don't-know.

Column labels are interleaved in the file for readability. The labels are interleaved after every 60 lines of text (including the labels themselves).

Figure 5.1 is an example of a standard simulation results file:

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>QB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>
5.2 Effect of the "SET" Command on the Simulation Results File

The "SET" command allows a user to redefine the symbols that are used to represent the logic-0, logic-1, and don't-know states. The "SET" command only changes the way in which the circuit output states are represented. Figure 5.2 is an example of a simulation results file when the user issued a "SET" command during a TSAR session and defined the logic value symbols in the following way:

```
Logic-0  0
Logic-1  1
Logic-3  XXXXX
```

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

A Standard Simulation Results File

Figure 5.1
A Simulation Results File Using the "SET" Command

Figure 5.2

5.3 Effect of the "MONITOR" Command on the Simulation Results File

The "MONITOR" command allows a user to observe the state of internal circuit points. Figure 5.3 is an example of the simulation results file when the user issued a "MONITOR" command during the TSAR session and added the following gates to the list of outputs: S, S2, R, R2.

<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>S</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>11000</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11000</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11001</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10001</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>11010</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>11010</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>11011</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>10011</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>11100</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11100</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>11101</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>10101</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>01110</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>01110</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>01111</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>01111</td>
<td>1</td>
</tr>
</tbody>
</table>

A Simulation Results File Using the "MONITOR" Command

Figure 5.3
5.4 **Effect of the "TRACE" Command on the Simulation Results File**

The "TRACE" command allows a user to observe transient states of the circuit outputs. Figure 5.4 is an example of the simulation results file when user issued a "TRACE 1" command.

![Simulation Results File Using the "TRACE 1" Command](image)

5.5 **Effect of the "SWITCH" Command on the Simulation Results File**

The "SWITCH" command allows a user to observe the states of the circuit outputs only when any of the outputs switches state and when the circuit is settled. Figure 5.5 is an example of the simulation results file when the user issued a "SWITCH" command during the TSAR session.
<table>
<thead>
<tr>
<th></th>
<th>Q</th>
<th>QB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>01</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>01</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>

A Simulation Results File Using the "SWITCH" Command

Figure 5.5
The table structure chosen for TSAR is described in this chapter. For a table-driven simulator, this structure is an important consideration with respect to the simulator's performance. The tables described here are appropriate given the following assumptions about the logic circuit:

1. Permissible logic elements are primary inputs, multiple-input single-output gates (AND, NAND, OR, NOR), wire-ANDed and wire-ORed nodes, and constant sources of logic-0 and logic-1.
2. An input to a gate has unity fanin.
3. The (single) output of a gate has multiple fanout.

Each element in the circuit has the following information stored in the memory-resident tables:

1. Element type
2. Number of inputs
3. Fanout number of its output
4. Pointer to its list of inputs and fanouts
5. List of inputs and fanouts
6. Current value
7. Propagation delays
8. Time at which it is scheduled to change values

Some supplementary information about the description of the circuit is also stored in the tables. This information includes:
1. A list of cross-coupled gate pairs (flip-flops). This list is used for critical race analysis.

2. A list of circuit outputs and user-specified monitor points. This list is used to report results. Note that circuit outputs are not considered to be special elements and need to be explicitly specified.

A map of the tables in main memory is shown in Figure 6.1. Each of the tables (lists) is detailed in the following sections.

---

Main Memory Map of the Circuit Description Tables

Figure 6.1

- 33 -
6.1 Topology Words

The list of topology words is made up of one integer word for each circuit element, where each word specifies the element's type, number of inputs, and fanout number. Four bits specify the type, 4 bits specify the number of inputs, and 6 bits specify the number of fanouts. Two bits remain unused presently. Figure 6.2 depicts a topology word for an element.

<table>
<thead>
<tr>
<th># Inputs</th>
<th>Fanout #</th>
<th>Element Type</th>
</tr>
</thead>
</table>

Element Type:

0 = Primary Input
1 = Battery
2 = AND
3 = NAND
4 = OR
5 = NOR

Topology Word as Stored in Main Memory

Figure 6.2

6.2 Pointers to the Inputs/Fanouts Lists

The list of pointers to the inputs/fanouts lists is made up of one integer word for each circuit element, where each word is an offset into the inputs/fanouts list. The offset points to the first input of the element. Figure 6.3 shows the list of pointers pointing to the inputs/fanouts lists of the circuit elements.
Pointers and Inputs/Fanouts Lists
as Stored in Main Memory

Figure 6.3
6.3 **Current Values**

The list of current values is made up of one integer word for each circuit element, where each word holds the current value (logic state) of the element. The permissible values are logic-0, logic-1, and "don't-know". Figure 6.4 shows the current value for an element.

```
<table>
<thead>
<tr>
<th>Logic Value</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0_8</td>
</tr>
<tr>
<td>1</td>
<td>-1_8</td>
</tr>
<tr>
<td>3 = don't-know</td>
<td>377_8</td>
</tr>
</tbody>
</table>
```

Figure 6.4

6.4 **Propagation Delays**

The list of propagation delays is made up of one integer word for each circuit element, where each word holds the nominal low-to-high and nominal high-to-low propagation delays of the element. Each delay is stored in a half-word. The permissible values for each delay are 0 through 255. Figure 6.5 shows the propagation delays for an element.
6.5 Schedule Table

The schedule table is made up of one integer word for each circuit element, where each word holds the time at which the element is scheduled to change value. This information is needed for spike analysis. Figure 6.6 shows an entry in the schedule table.

![Schedule Table Entry as Stored in Main Memory](image)

If \( \text{POINTER} = 0 \), then the element is not scheduled for a change in value.

If \( \text{POINTER} \neq 0 \), then \( \text{POINTER} \) points to the event that has already been scheduled.

6.6 Flip-Flop Pairs

The list of flip-flop pairs is made up of two integer words for each cross-coupled pair of gates in the circuit. Each pair of words holds the identification of the cross-coupled gates. Figure 6.7 details the list of flip-flop pairs.
6.7 Inputs/Fanouts

The list of inputs/fanouts is made up of a sublist for each circuit element. Each sublist is made up of one integer word for each input and fanout of the element. Each word contains the identification of the input or fanout. The length of each sublist is determined from the number of inputs and fanout number in the topology word for the element. (See Figure 6.3.)

6.8 Circuit Outputs and Monitor Points

The list of circuit outputs and monitor points is made up of one integer word for each element whose logic state is to be reported. Each word holds the identification of the element. Circuit outputs are not explicit elements and, therefore, need to be specified in this list. Figure 6.8 shows the list of circuit outputs and monitor points.
Circuit Outputs and Monitor Points as Stored in Main Memory

Figure 6.8
This chapter discusses how the logic circuit's response to new input stimuli is simulated. First, the algorithm is given which simulates the response of the circuit to new input stimuli until the circuit is settled. Then, an example is given which exercises the algorithm. Following the example is a description of the time queue mechanism and a discussion of the selective trace technique.

7.1 Algorithm to Simulate the Circuit's Response to an Input Stimulus

This section specifies the algorithm which "acts out" the circuit's response to new input stimuli.
DO UNTIL circuit is settled

IF oscillation criterion is met

THEN set oscillation-flag

Update current-time

FOR each event in the current-events-list

Replace the current-value by the scheduled-value

Reset the gate's schedule-table-entry

ENFOR

DO UNTIL end of the current-events-list

IELEM + fetch next gate from current-events-list

FOR each fanout-gate of IELEM

Compute new-value of fanout-gate

Current-value + fetch current value of fanout-gate

IF oscillation-flag is set

THEN

IF new-value differs from current-value

THEN override new-value with don't-know

IF new-value differs from current-value

THEN

IF fanout-gate is not already scheduled

THEN schedule this event

ELSE perform spike analysis

ENFOR

ENDDO

ENDDO
7.2 An Example Which Exercises the Algorithm Described in Section 7.1

Simulating the circuit response to new input stimuli boils down to simulating the response at each increment of time \( T_i \) from when the input stimuli are first applied \( T_0 \) to when the circuit settles. To simulate the behavior of the circuit at each time increment \( T_i \) where \( T_i \) represents some multiple of time units that has already expired, two basic tasks must be performed.

1. Update the current value of each element that was scheduled to change at \( T_i \).
2. For each element whose current value is updated, determine the effect on its fanout gates. For each fanout gate that will change value, schedule the change.

Of course, there are other clerical activities to perform.

An example follows which illustrates circuit simulation procedure described in section 7.1. The circuit is the same circuit shown in Figure 4.1 and is repeated here in Figure 7.1.

![NAND Flip-Flop Circuit Named FF](Figure 7.1)
The circuit description data is illustrated in Figure 7.2.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Type</th>
<th>Input Delay</th>
<th>High-to-Low Delay</th>
<th>High-to-Low Inputs List</th>
<th>Fanout List</th>
<th>Current Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Input</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Q</td>
<td>3</td>
</tr>
<tr>
<td>S2</td>
<td>Input</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Q</td>
<td>3</td>
</tr>
<tr>
<td>R</td>
<td>Input</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>QB</td>
<td>3</td>
</tr>
<tr>
<td>R2</td>
<td>Input</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>QB</td>
<td>3</td>
</tr>
<tr>
<td>Q</td>
<td>NAND</td>
<td>2</td>
<td>4</td>
<td>S, S2, QB</td>
<td>QB</td>
<td>3</td>
</tr>
<tr>
<td>QB</td>
<td>NAND</td>
<td>3</td>
<td>5</td>
<td>R, R2, Q</td>
<td>Q</td>
<td>3</td>
</tr>
</tbody>
</table>

Circuit Description Data for Figure 7.1

Let the input stimuli be S=S2=0 and R=R2=1.

Simulating at time $T_0$ (no time has expired) implies that the circuit is stable and new input stimuli are to be presented to the circuit. Implicitly, then, no events have been scheduled. The simulation begins by fetching the input stimuli and scheduling the circuit inputs whose value is changing. Circuit inputs are defaulted to have zero delay. Therefore, input changes would be scheduled at $T_0$. In this example, S and S2 change value from don't-know (3) to 0, and R and R2 change value from 3 to 1. So the events list at $T_0$ is:

$T_0$: $S$, 0  
$S2$, 0  
$R$, 1  
$R2$, 1

After the first pass through the procedure described in section 7.1, we have:

<table>
<thead>
<tr>
<th>Element</th>
<th>Current Value</th>
<th>Time Slot</th>
<th>Events List</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>$T_0$</td>
<td>S, 0</td>
</tr>
<tr>
<td>S2</td>
<td>0</td>
<td>$T_0$</td>
<td>S2, 0</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>$T_0$</td>
<td>R, 1</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>$T_0$</td>
<td>R2, 1</td>
</tr>
<tr>
<td>Q</td>
<td>3</td>
<td>$T_4$</td>
<td>Q, 1</td>
</tr>
</tbody>
</table>
Passing through the procedure again, we have

<table>
<thead>
<tr>
<th>Element</th>
<th>Current Value</th>
<th>Time Slot</th>
<th>Events List</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
<td>$T_7$</td>
<td>QB, 0</td>
</tr>
<tr>
<td>S2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QB</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Passing through the procedure again, we find no more changes occur and the circuit is settled.

In summary, an initial condition for the example circuit is shown in Figure 7.3.

![Initial Conditions of the NAND Flip-Flop](image)

The input stimuli $S=S2=0$, $R=R2=1$ are applied to the circuit as shown in Figure 7.4.

![Input Stimuli Are Applied to the NAND Flip-Flop](image)
And the circuit response simulated by TSAR is shown in Figure 7.5.

![Diagram of NAND Flip-Flop]

The Response of the NAND Flip-Flop to the Input Stimuli

Figure 7.5

This procedure is performed for each set of input stimuli.

7.3 **Time Flow Mechanism**

The time flow mechanism is defined to be the algorithm which drives the simulation through the time domain\(^\text{13}\). The mechanism used in TSAR is designed to be a hybrid of the "fixed time increment" mechanism and the "next event" mechanism\(^\text{14}\). Presently the fixed time increment mechanism is implemented. The next event mechanism does not need to be implemented until the limit on the propagation delay for an element is increased.

The fixed time increment mechanism allocates \(N\) slots to represent \(N\) sequential units of time. For example, in TSAR \(N = 256\). Each time slot points to an unordered list of events that are scheduled to occur at the time represented by that slot. An event here is


defined to be an element switching to a new value. The event is represented by an element identification and the logic value to which it must switch. Figure 7.6 illustrates this time queue.

Data Structure for the Fixed Increment

Time Flow Mechanism

Figure 7.6
In TSAR, the queue is actually circular and each slot represents \( T_1 \) modulated by 256. To calculate \( T_1 \) for actual circuit settling time, \( T_1 \) must be increased by a term given by \( 256 \times M \), where \( M \) is the number of times a cycle through the queue has been completed. Because the maximum propagation delay is 255, a queue length of 256 handles all possible scheduling of events. For example, if the current time in the simulation time domain is \( T_2 \), no element could possibly switch value later than \( T_{257} \) which corresponds to time slot \( T_1 \) (257 modulo 256).

If the maximum propagation delay is ever extended to be greater than the length of the time queue, then the next event time flow mechanism must be implemented. The mechanism maintains an ordered list of events that are scheduled beyond the time represented by the fixed increment time queue. The ordering is based on increasing time increments. At each new increment of time then, not only are the events carried out that are pointed to by a slot in the fixed increment time queue, but also, those events are carried out in the ordered next events list that are scheduled for the appropriate increment of time.

7.4 Selective Trace

Selective trace is a technique commonly used in logic simulators. The technique simply follows circuit activity (only switching gates) rather than blindly performing the algorithm described in section 7.1 on all the circuit elements all of the time. Specifically,
when an element switches value, only its fanout gates that consequently change value are scheduled.

This technique greatly reduces execution time of the simulation program.
CHAPTER 8
PROGRAM PERFORMANCE

This chapter gives real elapsed time in simulating several circuits. The data indicates how well the program performs.

The timing tests were done on three circuits of various sizes. Input and output was done via mag tape. TSAR was the only program scheduled and running under the RTE-III operating system. The total simulation time is the real time elapsed from when the RUN command (see section 4.4) was issued until the entire sequence of input stimuli was applied to the circuit. It is important to note that this elapsed time includes writing the simulation results to mag tape. Figure 8.1 shows the results of the timing tests.

Following are the explanations of the column headings in Figure 8.1. The total number of elements is equal to the number of circuit inputs and the number of gates in the circuit (see section 4.1). The input sequence length is the total number of input patterns in the sequence of input stimuli (see section 4.2). The maximum propagation delay is the largest number of time units it took for the circuit to settle in response to the sequence of input stimuli. The elapsed time, again, is the number of seconds elapsed from the time the RUN command was issued by the user until another command prompt was issued by TSAR.
<table>
<thead>
<tr>
<th>Total Elements</th>
<th>Input Sequence Length</th>
<th>Maximum Propagation Delay</th>
<th>Elapsed Time (Sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>37</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>339</td>
<td>129</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>585</td>
<td>258</td>
<td>13</td>
<td>75</td>
</tr>
</tbody>
</table>

Elapsed Time Required to Simulate Some Sample Logic Circuits

Figure 8.1
CHAPTER 9

PROGRAM LIMITS

This chapter documents the limits imposed by the software on the circuit itself and on the sequence of input stimuli applied to the circuit.

9.1 Circuit Limitations

Presently, the largest circuit that can be simulated is limited to about 400 gates and circuit inputs. This limit can be increased to about 3000 gates and circuit inputs if the Extended Memory Area (EMA) under HP RTE-IV (see section 2.3) is used. Presently, usage of EMA is not implemented.

Because of the data packing in an element's topology word (see section 6.1), the number of element types is 8, the number of fanouts per element is 63, and the number of inputs per element is 15. Any of these three limits may be extended because there are two unused bits in the topology word.

The high-to-low and low-to-high propagation delays may range between 0 and 255 (see section 6.4). These limits may be extended by storing each in a computer word, for example. Of course, the amount of storage needed doubles. For a 1000-gate circuit, 2000 words would be needed to store propagation delays. The current implementation would require 1000 words.

The number of flip-flop pairs is not explicitly limited. Implicitly, the number is limited by the amount of storage for the entire circuit description.
The number of circuit inputs is limited to 77. TSAR assumes each input stimuli to be contained in a card image. The stimuli are specified by T'<input stimuli>'. The three characters, "T" and two ""s, leave 77 characters left in a card image to specify the input stimuli.

The total number of circuit outputs and monitor points is limited to 50 in the present implementation. This limit is readily extended by changing a parameter in the program.

9.2 Input Stimuli Limitations

Each input stimulus is limited to three logic values: 0, 1, and 3 (don't-know).

The sequence of input stimuli is limited in the present implementation by the amount of disc space available. The entire sequence is stored in a disc file. Blocks of stimuli are read into main memory as needed.
CHAPTER 10

FUTURE IMPROVEMENTS

This chapter outlines three major improvements that could be made in TSAR. Ordered by the author's priorities, the improvements are:

1. Simulate larger circuits.
2. Implement a clocking mechanism for input stimuli.
3. Simulate singly occurring stuck-at-0 and stuck-at-1 faults.

10.1 Larger Circuits

Foremost on the list of improvements is to make use of the Extended Memory Area (EMA) on the HP 21M-series minicomputers. Using EMA would allow circuits of up to 3000 gates to be simulated.

10.2 Clocking Capability

The ability to describe clock pulses in a user command language makes the simulator easier to use. Presently, clock pulses must be described explicitly in the sequence of input stimuli. For example, T'1...', T'0...', T'1...' defines the following pulse:

```
1 ______
\       |
0 ______
```

Whereas, a command like "PULSE 1,0" could describe the pulse more easily. Moreover, it is more readily edited by a user.
10.3 **Fault Simulation**

The ability to simulate faults in a circuit is essential in predicting how many faults a sequence of input stimuli can detect. TSAR can readily be extended to simulate singly occurring stuck-at-0 and stuck-at-1 faults. The procedures needed to be implemented are fault list generation, fault insertion, and fault detection (see section 3.2).
BIBLIOGRAPHY


VITA

Brenda M. Csencsits was born to Kathleen and Bert Csencsits on December 16, 1951, in Walnutport, Pennsylvania. She received a B.A. in Mathematics from Shippensburg State College in Pennsylvania. Since 1973, she has been working at Bell Laboratories in Allentown, Pennsylvania. Her work there involves developing computer aids for integrated circuit design.