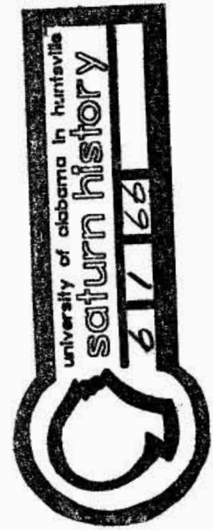

SATURN HISTORY DOCUMENT
University of Alabama Research Institute
History of Science & Technology Group

Date ---------- Doc. No. --------

IBM No.: 66-825-1992

XI, 16

DESIGN AND USE OF FAULT SIMULATION FOR SATURN COMPUTER DESIGN

by

F. Hardie
R. Suhocki

June 1966

IBM *Federal Systems Division, Electronics Systems Center, Owego, New York*

CONTENTS

PART I - SIMULATOR

INTRODUCTION 1

THE PROGRAM SYSTEM 1

- Data Collection 2
- Logic Selection 2
- Fault Injection 3
- Intermittent Fault Analysis 4
- Logic Compilation 5
- Simulation 7
- Race Monitoring Feature 8
- Saturn Program Execution Trace 9
- Functional Memory Dumps 9
- Executive Program Summary 9
- Simulator Output 9
- Program System Modularity 11
- Simulator Running Time 12

PART II - SIMULATION

INTRODUCTION 13

SIMPLEX SIMULATION 13

- Design Verification 14
- Test Program Evaluation 14
- Test Point Catalog Generation 15

REDUNDANT SIMULATION 15

- Detection System Evaluation 15
- Disagreement Detector Placement 16
- Voter Placement 20
- Intermittent Failure Analysis 20

ACKNOWLEDGEMENTS 21

REFERENCES 22

DESIGN AND USE OF FAULT SIMULATION FOR SATURN COMPUTER DESIGN

by

F. Hardie
R. Suhocki

International Business Machines Corporation
Federal Systems Division
Electronics Systems Center
Owego, New York

PART I - SIMULATOR

Introduction

The Saturn Fault Simulator is a system of programs to be executed on an IBM 7090 computer. The objectives of this simulator were:

- Verify the logic design of the Saturn computer
- Analyze the effects of solid plus intermittent faults
- Evaluate the effectiveness of the Saturn Diagnostic programs through fault simulation
- Evaluate changes in design before commitment to hardware.

The significant characteristics of this simulator are:

- Full Central Processing Unit simulation while containing in one 32K memory the complete compiled logic simulator and a simulated Saturn memory module plus interface data
- Fault simulation capability, including single or multiple, solid or intermittent faults
- Approximately 2,000 Saturn clock times are simulated per minute
- User-specified output options plus an output editing program which controls the mass of simulation output
- Solid Logic Design Automation system used to define the logic to the simulator. Simulation is carried out at the basic AND-OR-INVERT level rather than at a functional level.

To achieve a practical simulation speed, the following was accomplished:

- A compiled simulator rather than an interpretive type was chosen. This compiled simulator is completely contained in core storage
- A technique called "Parallel Error Simulation" allowed simultaneous normal plus 33 fault simulations at no decrease in speed
- A technique called "Stimulus Bypassing" was developed. Essentially, this means that groups of compiled 7090 instructions will not be executed if specific conditions exist
- Program system modularity was established to allow the flexibility of using only the routines which are necessary in a given run.

The system allows the user to specify up to 100 source nets or test points which will be monitored during simulation. The binary values for these points will be printed out according to a variety of options.

This paper consists of two parts. Part I describes the programs, while Part II describes the application of the simulator. The paper describes the programs in the same order that they would normally be executed, as shown in Figure 1.

The Program System

The following features of the system will be described:

- Data Collection (SLDA)
- Logic Selection
- Fault Injection
- Intermittent Fault Analysis

- Logic Compilation
- Simulation
- Race Monitoring Feature
- Saturn Program Execution Trace
- Functional Memory Dumps
- Executive Program Summary
- Simulator Output
- Program System Modularity
- Simulator Running Time.

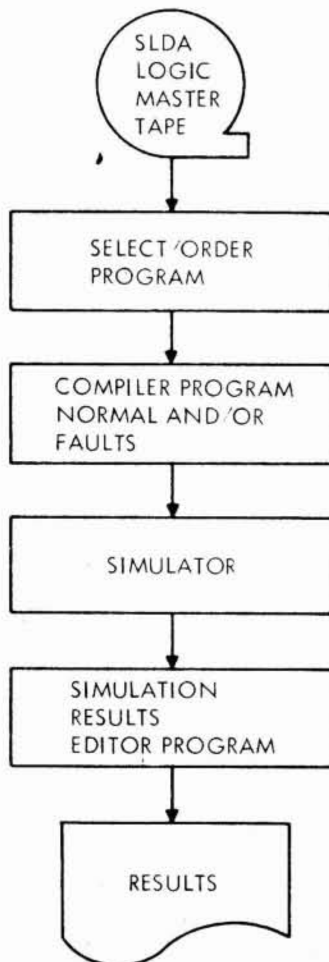

Figure 1. General Flow of the Saturn Fault Simulator

Data Collection. The SLDA system is used to record the logical design of the Saturn computer.

While this paper does not intend to delve into a detailed description of SLDA, the general flow, as it pertains to the Saturn Fault Simulator, is discussed.

The logic is first manually drawn on SLDA sketch sheets. Keypunch operators then convert the sketch sheet information into SLDA punched cards. Using these cards, SLDA produces a Logic Master Tape (LMT) which now serves as the basic source of logic input to the Saturn Fault Simulator. SLDA also produces drawings called Automated Logic Diagrams (ALD) which portray the logic as it is defined on the LMT. Figure 2 shows the data collection system.

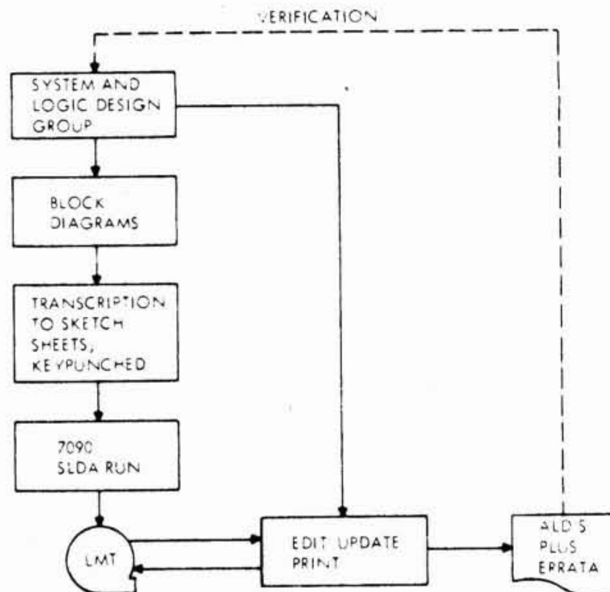

Figure 2. Data Collection System

Logic Selection. The "Select" program is that part of the simulation system which interfaces with SLDA. "Select" has the capability to read the LMT, extract user specified logic blocks from it, and prepare these blocks for logic compilation. A key problem in logic simulation is attacked by our Select procedure: i.e., the logic must be simulated in the same fashion, or order, that the actual information is propagated in the Saturn computer. If this logical ordering is not achieved the simulator would have to loop, or repeatedly simulate the logic while holding time constant, until correct propagation has occurred. "Looping" the complete Saturn computer logic would greatly increase the IBM 7090 Computer running time, which increases the cost of using the simulator. To attack this problem the logic is "selected" from the LMT in a specific sequence. Block functions

are selected in the same order that they would perform their functions in the computer. At present, this a man-machine procedure. The correct order for the logic block functions is manually determined and specified to the Select program, which then performs the desired logic selection and logic ordering. The general order of logic selection for the Saturn computer is:

- Timing logic
- Combinational logic (AND, OR, INVERT)
- Sequential logic (latches, tratches, etc.)

The end result of the Select procedure is a tape which contains all of the logical data necessary for a properly ordered, compiled simulator.

Fault Injection. One reason for the success of any computer logic fault simulator is the ease with which faults may be specified by the user. A fault may be thought of as the transformation of one logical function into another. A fault could be considered to be a terminal node of a logic block stuck at "logical one" or "logical zero". If the user were forced to re-define the logic to simulate a fault condition, he could be reluctant to make extensive use of a fault simulator. Consider a complex Boolean function logic description. The simulation of a fault would require that the Boolean equations be studied and changed to reflect the desired fault condition. In contrast, the approach taken in the Saturn Fault Simulator is to maintain a constant, fixed logic description (the LMT). Faults are then specified by the user in a very simple manner. He merely specifies:

- The ALD page upon which the failed terminal is shown
- The logic block serial number of the failed terminal
- The line to or from the logic block which represents the failed terminal
- The logical type of fault (0 or 1).

Figure 3 portrays a typical ALD logic block and the desired fault situation.

The characteristics of this method of fault simulation can be summed up as follows:

- The fault injection procedure is separated from the logic description part of the system. A complex logic description does not have to be redefined by the user.

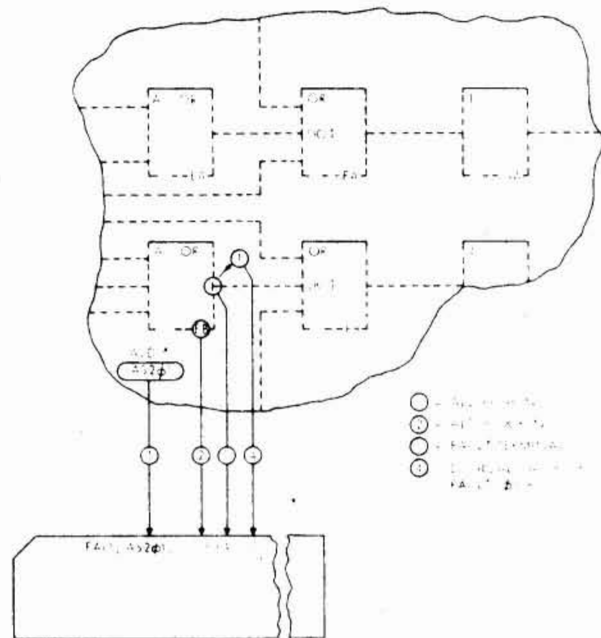

Figure 3. Fault Injection Example

- Fault injection is separated from the logic compiler program. Thus the same simulator can be used for either normal design analysis or fault simulation experiments.
- The faults to be simulated can be the terminals of any logic block on an ALD, stuck to either logical one or logical zero.
- Intermittent fault analysis is performed at simulator execution time. Since the intermittent fault can be considered to be a special case of a solid fault, intermittent fault specification is the same as solid fault specification as far as the user is concerned.
- Single faults or multiple faults can be simulated. Up to 33 single faults or 33 groups of multiple faults (up to 25 faults per group), plus the normal machine, can be simulated at one time in the IBM 7090 Computer, with no increase in running time over the single-fault simulation case. This is possible because of a technique used in the programs called Parallel Error Simulation.

In paralled error simulation, each bit position in the 36-bit 7090 computer word can be used to contain the binary value of a specified fault or a specified group of multiple faults. In actual practice, the Saturn design dictated that three bits be reserved for the normal (unfailed) machine and the remaining 33 bits used for fault simulation. The mechanics of parallel simulation can be illustrated by an example of single-fault simulation.

Consider a three-input AND logic block. It is desired to "fail" the first input to "1". It is also desired to "fail" the second input to "1", yet keep these faults independent from each other. They are single faults in this example, not multiple faults. The 7090 FAP (FORTRAN Assembler Program) instructions for this block would be as follows:

CAL	INPUT 1	Pick up first input
ORA	MASK 1	"FAIL" it
SLW	OUT	Save "FAILED" input
CAL	INPUT 2	Pick up second input
ORA	MASK 2	"FAIL" it
ANA	INPUT 3	Perform "AND" function
ANS	OUT	Perform final "AND" function

In this example, assume that the three inputs to the block have the binary values 011, that is:

INPUT 1 (Unfailed) = 0

INPUT 2 (Unfailed) = 1

INPUT 3 (Unfailed) = 1

Since 36 bits are used for each input in our example, the following bit assignment is made (numbering bits 1 through 36):

BITS 1, 2, 3 --- normal machine value

BIT 4 --- INPUT 1 failed to 1

BIT 5 --- INPUT 2 failed to 1

Thus, in the example, the following values exist prior to simulation:

INPUT 1 - 000000 --- 0

INPUT 2 - 111111 --- 1

INPUT 3 - 111111 --- 1

MASK 1 - 000010 --- 0

MASK 2 - 000010 --- 0

OUT - XXXX --- X (Don't care)

If we carry out the simulation of the failed logic, showing the values of the accumulator and "OUT", we have:

Compilation	Accumulator	Out
CAL INPUT 1	00000-------0	XXX ---X
ORA MASK 1	00010-------0	XXX ---X
SLW OUT	00010-------0	00010---0
CAL INPUT 2	11111-------1	00010---0
ORA MASK 2	11111-------1	00010---0
ANA INPUT 3	11111-------1	00010---0
ANS OUT	11111-------1	00010---0

Upon examination of the final value of location "OUT", the following conclusions can be drawn:

- The normal machine value, as shown by bits 1, 2, 3, is still zero, as it should be
- INPUT 1 failed to "1", results in the final block value of "1", differing from the normal machine value. BIT 4 is a "1"
- INPUT 2 failed to "1" does not cause the failed value to differ from the normal value. BIT 5 equals BIT 1.

Although parallel simulation necessitates a certain amount of bit manipulation in the simulator, this added work is far exceeded in value by the resulting reduction in running time of the IBM 7090 Computer.

The mechanics of multiple-fault simulation are similar to single-fault simulation. The difference is that for multiple faults a group of faults is forced to affect only one bit position in the computer word rather than a unique bit position for each fault. The user defines, through control cards, whether a single or multiple faults are to be simulated. In any given 7090 Computer run, several single or multiple faults may be specified.

Intermittent Fault Analysis. Two different techniques were used for intermittent fault simulation. Initially, it was decided to have the user define the behavior of the intermittent faults. This technique was followed by a second in which the user did not define the characteristics, but merely followed the fault injection procedure

previously discussed. Each of these two techniques shall now be presented.

In the first procedure, the user defined the following:

- The logic block fault terminal
- The time for fault injection, which is defined by the Saturn operation code and clock time
- The time for fault removal (normalization) also defined by Saturn operation code and clock time.

During the simulation of the Saturn test program, the desired faults were either injected or removed. This procedure depended upon the user's knowledge about the behavior of an intermittent fault, its effect upon the computer, and its detection by the diagnostic programs. Rather than having the user determine these factors, it was decided to make the Saturn Fault Simulator provide this information.

In the second procedure, the user had only to define the logic block fault terminals. The time of injection or removal was not specified by the user.

The simulator assumed that a specified fault was "solid" until detected. Detection caused certain counters in the simulator (not hardware) to be incremented. The simulator presented several characteristics about the intermittent fault including:

- The number of times the fault affected the logic
- The percent of cycle time during which the logic was affected
- The number of times the fault was detected
- The percent of occurrences which were detected
- The times of detection (in microseconds)
- The interval between detections (in microseconds).

This type of statistical approach enabled the user to study such problems as:

- How many times does a given fault actually affect the behavior of the computer?
- How sensitive is the computer to specific fault conditions?
- Is the effect of a given fault "latched up" or is the effect transient in nature?
- Of the faults which affect the behavior of the computer, how many are detected by the diagnostic programs?
- How long, in microseconds, must an intermittent fault remain "solid" before being detected by a given diagnostic program?
- What are the probabilities of detection for a given intermittent fault as we allow its duration, or "period of solidity", to vary?

In summation, what is an intermittent fault, how does it affect the computer, can it be detected and diagnosed? Information about the actual intermittent fault simulation and analysis can be found in Part II of this paper.

Logic Compilation. Up to this point, Data Collection, Logic Selection, and Fault Injection have been presented. These three subsystems supply all of the logical information necessary to generate a compiled logic fault or normal simulator. The Compiler itself consists of a main program plus a series of subroutines, each subroutine programmed to compile a specific type of logic block. Different subroutines are allowed to exist for the same type of logic block, thus allowing the user to experiment with different logical models. The Compiler has the following characteristics:

- It interfaces with a master logic input tape, automatically produced by the preceding system programs. Thus, any changes in the system logic can quickly and automatically be incorporated in the compiled logic simulator.
- An actual compiled simulator is produced, consisting of 7090 FAP instructions ready for execution. Efficient use of 7090 FAP instructions plus data results in less computer cost at simulator execution time.
- Fault models are easily compiled using the subroutine approach.

Figure 4 shows a simplified portion of a compiler subroutine flow chart. Note that the compiler approach taken in the subroutine is very straightforward.

The simulation of a type of logic block called

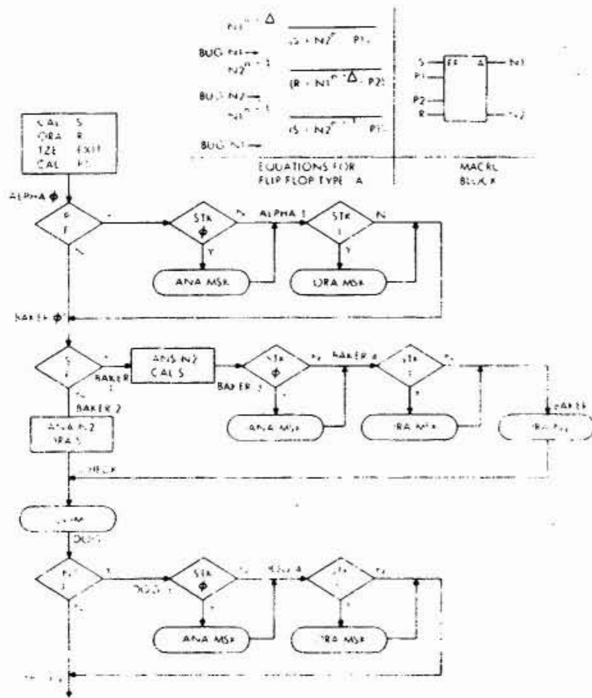

Figure 4. Portion of Compiler Subroutine Flow Chart for Type A FF

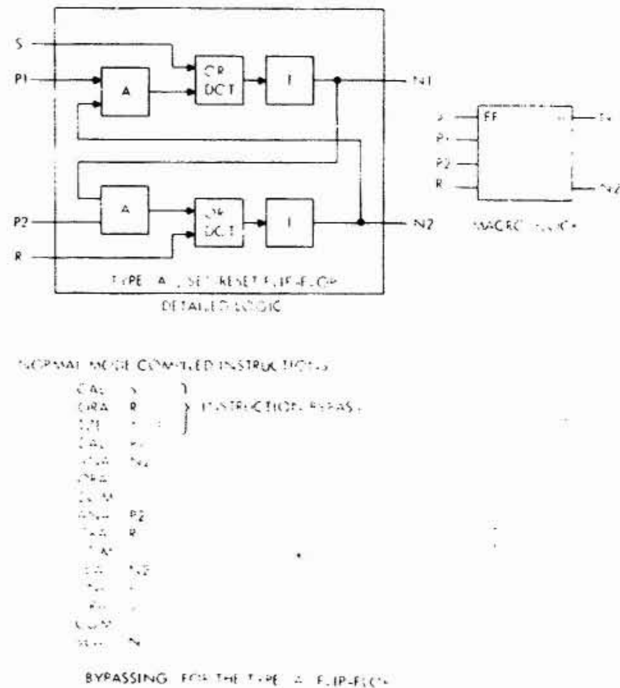

Figure 5. "Bypassing" for the Type "A" Flip-Flop

the "macro block" was implemented. Basically, a macro block consists of the AND-OR-INVERT (A-O-I) logic which makes up a specific function, such as a latch. Rather than define the individual A-O-I logic blocks on the ALD's, the macro block is used. Several advantages result from the use of macro blocks, including:

- ALD's which are easier to read
- Fewer errors in the logic definition phase since fewer blocks are defined
- Faster simulation speed since the macro block makes possible a simulation technique called "Instruction Bypassing"
- Feedback loops within latch configuration can be isolated within a macro block. Figure 4 shows a flip-flop macro block and its internal logic.

Instruction bypassing can be defined as the skipping of specific computer instructions during simulation. If the simulator can determine that a logic block cannot possibly change state during a simulation pass, then there is no need to execute the 7090 Computer instructions which simulate that particular logic block. Consider the flip-flop shown in Figure 5. If neither a set nor a reset signal is present, the flip-flop will not change state. If a macro block can change state only at specific clock time, it is necessary to simulate the block only at that particular time. It is obvious that 7090 Computer instructions must be added to the basic compiled logic simulator to test for these bypass conditions. It is not economically feasible to add these codes for the individual A-O-I logic blocks, but it is feasible to do so for the macro block. For a set-reset flip-flop, the bypass requires three additional FAP operations, namely:

CAL	SET VALUE
ORA	RESET VALUE
TZE	BYPASS

This will result in the bypassing of 12 FAP operations when neither the set nor the reset is present. Since most flip-flops are inactive at any given time during the simulation, the by-pass results in a significant saving in execution time. The normal set-reset case just presented illustrates the minimum saving. If faults had been specified for the flip-flop, more instructions would be bypassed. The Set-Reset flip-flop is one of the most simple sequential models simulated. The more complex models result in an even greater saving in execution time due to instruction bypassing.

The final product of the Compiler is the compiled logic simulator. It consists of three major sections:

- The compiled FAP instructions
- A table reserved for the logic block values
- A master list stating which location in the table of logic block values represents each logic block output terminal.

A compiled FAP instruction consists of a 7090 Computer operation plus an address. The address refers to a location in the table of logic block values. These addresses are not converted to absolute 7090 Computer locations until simulator execution time. Thus, the compiled simulator and the table of logic block values can be located anywhere in the 7090 Computer core storage at execution time. This flexibility is noteworthy since it allows fewer fixed table sizes at execution time, thus extending the capacity of the simulator through complete usage of the 7090 Computer's core storage. Since dynamic storage allocation is used when it is practical, it is difficult to determine the largest size logic network which can be simulated. A logic network will be compiled and simulated if it can be contained by the system in the 7090 Computer core storage. If the logic exceeded the 32K memory, other techniques would have to be used. Techniques such as packing/unpacking the compiled simulator, or using tapes for additional storage, or a very sophisticated functional-logical approach would have to be programmed. It is estimated that execution time on the 7090 Computer would increase by at least a factor of 10 if core storage were inadequate.

So far, the described programs have accomplished the following:

- Converted the designer's logic sketch sheets into a computerized language
- Selected the logic to be simulated in such a way as to provide efficient, loop free simulation
- Compiled a model of the logic in the 7090 FAP language, with or without user specified faults.

Simulation. To execute the compiled model, the user must specify several factors to the Executive program. These are listed below, but are not discussed in detail:

- Mode - either the NORMAL or a FAILED mode will be simulated
- Simulation Limit - The user specifies the maximum number of Saturn instruction cycles to be simulated. Termination of the simulation will occur either when this limit is reached or when an "END" operation code is reached in the Saturn program being simulated.
- Output Options - The following options are available:
  - Display the binary values of all specified output terminals at each interval of Saturn clock time.
  - Display the binary values of all specified output terminals only when a terminal changes value.
  - Display the Saturn computer disagreement detectors.
- Output Terminals - The user specifies up to 100 output terminals which will be displayed. They are specified by their ALD net names (page, block and terminal number)
- Output Registers - The user can specify output registers as well as individual output terminals, since a register can be thought of as a collection of specific output terminals. The limitation is that only 100 binary values can be printed as simulator output, so no more than 100 terminals (total) should be contained in the specified registers. The collection of terminals which constitute a register is defined once at compilation time, rather than execution time, since this definition is fixed by the design.
- Output Times - If output options are not specified, the user can specify the exact Saturn clock times at which he wants the output terminals displayed. Display occurs each time a specified clock time is passed in the simulation
- Logic Initialization - The user can initialize any logic block output terminal or any interface line (I O, control, power, etc.) by specifying the ALD net name and the desired initial value (logical 1 or 0)
- Simulated Saturn Program Flow - Normally, a fault being simulated is not

allowed to affect the value of the Memory Address Register. Thus the program flow determined by the good, unfailed logic, is followed. If the user desires, a specified fault will be allowed to set the value of the Memory Address Register. This procedure exists because parallel simulation makes it possible to simulate many faults in one 7090 Computer execution, but for one Saturn program and one program flow. To follow all possible program flows for all faults would revert the simulation procedure back to the one fault per 7090 Computer run technique.

- The Saturn Program - The operational or diagnostic program to be loaded into the simulated Saturn memory was coded by the user in his assembly language. A memory loader existed in the system, performing the task of converting the Saturn operation codes and data into the proper memory bit patterns and storing these patterns in the correct simulated memory loactions. Checking for illegal Saturn operation codes was performed.

Certain areas with the Saturn computer did not lend themselves easily to logic simulation. These included the decode and select circuits for the simulated memory module. For purposes of fault simulation and central processing unit (CPU) design verification, it is necessary to simulate the logic up to a well defined memory interface. The memory address register (MAR) and the memory buffer register (MBR) plus the read/write signals constitute the interface. The binary values for the interface emanate from the simulated logic during a write procedure. The executive program then functionally decodes the MAR and stores the contents of the MBR in the addressed location within the block of 7090 Computer storage defined to be the "memory" of the Saturn computer. A Read operation is the converse in that the executive program decodes the MAR and moves the binary contents of the addressed location into the MBR, from which point logic simulation propagates it through the central processing unit. Figure 6 illustrates the memory interface as it appears to the simulator.

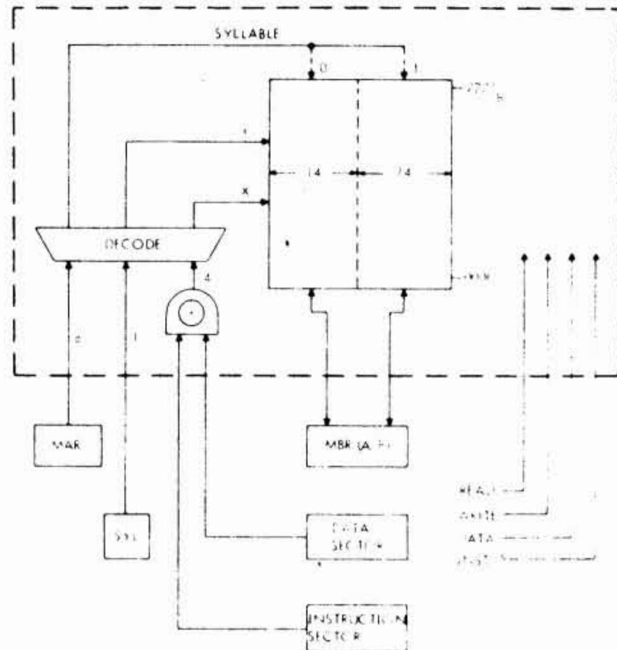

Figure 6. Memory Module Interface

Race Monitoring Feature. A "race" situation exists in a logic design if one or more state determining signal paths are of equal "length" and are excited simultaneously. The final circuit state could be indeterminate, since small defferences in the path and element delays can change the actual final value. Since the Saturn computer design employs disagreement detectors which monitor circuit states, a false error indication can result from a non-critical race. To reduce or eliminate circuitry which would cause these race conditions the simulator was modified in the following manner:

- At Compiler time, a table containing the ALD net names of the sequential elements in the logic (flip-flop traches) was constructed
- Also in this table the indices to the locations in the block value table for the SET and RESET output values for each sequential block were saved. This table of names and indices is passed on to the Executive program along with the compiled simulator.

The Executive program uses this table for race monitoring in the following manner. After each simulation pass of the logic, the Executive program examines the block value locations indexed by each pair of SET-RESET indices. If it finds that both SET and RESET equal "logical zero" it produces a diagnostic message for the user, stating that: "Flip-flop 0-0 state occurs at time _X_, instruction _Y_, page _A_, block _B_, environment _N_." where:

X is the Saturn clock time, such as "A01W."

Y is the location in simulated Saturn memory for the Saturn op code being executed, as "23".

A is the ALD page name on which the flip-flop can be located, as "AA201".

B is the block designation of the flip-flop as it appears on the previously defined page, as "BA".

N is the fault number, but since multiple faults can be simulated, we prefer to call it the environment number, as "5". Thus, the effect of a single fault or the effect of a group of faults or the fault free normal behavior of the logic could have caused the race condition.

Note that this 0 - 0 flip-flop condition was caused by both the SET and the RESET input signals coming up to a logical "1" value at the same time. If only one signal, either the SET or the RESET, then returns to a 0 state, the flip-flop will stablize to a predictable output state, but if both signals return to a "0" state the flip-flop final state cannot be computed by the simulator. Either the true output will be a "1" and the complement output a "0", or vice versa. The simulator brings this situation to the attention of the user and he decides if it is critical. If he decides that the race situation is not critical, he can suppress the race monitoring for particular flip-flops by punching the proper execute control cards.

Saturn Program Execution Trace. The Executive program decodes the value of the MAR and prints out every time a Saturn instruction is read from memory. The Saturn address is broken down into three subfields: the memory sector, the location within the sector, and the syllable within the location. The cause of the instruction fetch, namely the good machine or a particular fault environment is also printed out. Thus, the user knows the flow path of the executed Saturn program and whether it was caused by a particular fault or the normal machine. In most simulation runs, only the good machine value of the MAR is actually used to determine the "next instruction address", but fault-caused values of the MAR are printed out since this is a form of fault detection. Figure 7 illustrates a memory trace.

Functional Memory Dumps. In addition to being used as a logic design and evaluation tool, the simulator could aid in the debugging of Saturn operational or diagnostic programs. It was not intended that the simulator be extensively used for program debugging since its 7090 Computer execution time could be much higher than a functional operation code simulator (a normal difference between a compiled logic simulator and an operation code simulator). However, fault conditions will present an environment for the Saturn program which is not achievable in most operation code simulators. Also, since the normal behavior of a Saturn program is not checked out for every possible data combination, the possibility of a program "bug" does exist. To aid in the checking of the Saturn program, in addition to checking the logic, the Executive programs dumps the simulated Saturn memory before and after the simulation of a Saturn program. Thus the user can examine the results of Saturn program execution by storing the results in memory and examining the final memory dump.

Executive Program Summary. As stated previously, several programs in the system have produced a compiled logic simulator. It is up to the user to control this simulator to obtain useful information. He does this by punching the executive program control cards, which will specify the terminals to be displayed, the rate or time of display, the number of Saturn operation codes to be simulated, the Saturn program to be executed, the logic block initial conditions, and in general, all conditions which could vary and still allow use of the same compiled simulator. A limitation to be remembered is that up to 100 terminals in the logic may be displayed during one 7090 Computer run. A very general picture of the simulation flow at execution time is shown in Figure 8.

Simulator Output. As with any program excessive computer input output operations, including card reading, on-line printing, tape reading and writing, etc., tends to contribute significantly to computer running time. The techniques already stated have tremendously helped to minimize the actual in-core simulation time. To minimize the time necessary to present the results to the user the following approach was taken:

- The Executive program writes a high density, binary output tape. This tape contains the values of the user specified logic block terminals at user specified output times. Prior to execution of the compiled simulator, the Executive program reads the user supplied output control cards and sets up a table containing the addresses of the locations in the 7090 Computer storage which contain the logic block values for the specified output terminals.

- During execution, this table is used to quickly locate the output terminal values and move these values to one preassigned output buffer. This done only when the Executive program decides that it is time for a user specified output.

- The Executive program then writes the output value table on tape, one binary record, with no conversion to BCD.

MACH	SECT	ADDR	SYLL	MACH	SECT	ADDR	SYLL	MACH	SECT	ADDR	SYLL	MACH	SECT	ADDR	SYLL	MACH	SECT	ADDR	SYLL
/ 00	00	001	0																
/ 00	00	002	0																
/ 00	00	003	0																
/ 00	00	004	0																
/ 00	00	005	0																
/ 00	00	006	0	/ 23	00	005	0												
/ 00	00	007	0	/ 25	00	006	0												
/ 00	00	011	0																
/ 00	00	012	0																
/ 00	00	013	0	/ 22	00	012	0												
/ 00	00	014	0																
/ 00	00	015	0																
/ 00	00	016	0																
/ 00	00	040	1																
/ 00	01	315	0																
/ 00	01	316	0																
/ 00	01	317	0	/ 02	01	370	0	/ 05	01	370	0	/ 09	01	370	0	/ 28	01	370	0
				/ 29	01	370	0	/ 30	01	370	0	/ 33	01	370	0				
/ 00	01	320	0																
/ 00	01	277	1																
/ 00	01	300	1																
/ 00	01	301	1																
/ 00	01	302	1	/ 03	00	000	0	/ 04	00	000	0	/ 16	00	000	0	/ 27	00	000	0
/ 00	01	303	1																
/ 00	01	304	1																
/ 00	01	305	1	/ 01	01	370	0	/ 07	01	370	0	/ 08	01	370	0	/ 15	01	370	0
				/ 17	01	370	0	/ 19	01	370	0	/ 20	01	370	0	/ 21	01	370	0
/ 00	01	306	1																
/ 00	01	307	1																
/ 00	06	120	0																
/ 00	06	121	0																
/ 00	06	122	0																
/ 00	06	123	0																

Figure 7. Memory Trace

- An Edit program follows the complete simulation run. The user specifies the faults (actually the environment numbers) for which the binary simulation results are to be converted to BCD, set up in the proper format, and written on another tape ready for printing.

The advantages of this approach are:

- More 7090 Computer locations are available for actual simulation since the programs which convert the binary logic block values to BCD, in a format acceptable to the user, are not in the 7090 Computer at simulation time.
- Output format flexibility is achieved. Since the Edit program can also use all of the 7090 Computer storage, it can allow more format specification by the user. The output terminal names, which are either signal names or ALD names, plus the column assigned to that terminal in the output format, are user specified. Thus the user names and groups the terminal values in whatever format is most useful to him.
- The user can specify that part, or all, of the environments to be printed. The input tape to the Edit program is saved, thus the Edit program can be used to print more copies, or to print only selected environments.

Figure 9 shows a sample output after Editing. The environment number, plus a list of the faults in that environment, are shown at the top of the

output. The output terminals were defined by the user in two ways:

- As ALD names (page, block, output line number)
- As Signal names (EP01, EP02, etc.).

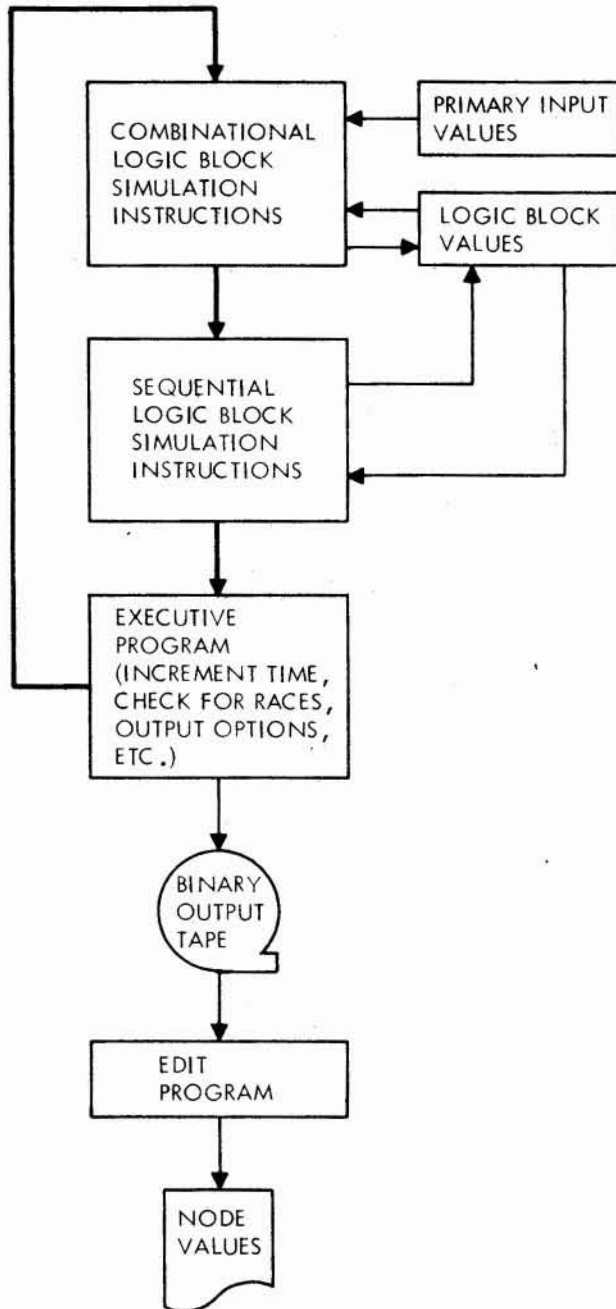

Figure 8. Simulation Flow

The ALD names are essential in order that the Executive program can monitor the proper logic block values. However, signal names could have more meaning to the user. If he specifies both the ALD name and the signal name, the Edit program will print the signal name. The information on the left of the page shows the Saturn clock times (Phase, Bit Time, Clock) at which the output terminal values were monitored. The binary values in the center of the page are read from top to bottom, showing the changing values of the monitored terminals. The information on the extreme right of the page is the memory address register value, thus showing the location of the Saturn operation code which was being executed at that time.

Program System Modularity. Figure 10 shows the system of simulator programs. Since the simulator is composed of distinct programs, each having a unique purpose, it is recommended that the simulator be used in a modular fashion. Consider the following aspects:

- The SLDA system provides the logic description necessary for compilation and simulation. It is necessary to run the "SELECT" program only when the logic, as defined on the LMT, is changed. Therefore, the output tape from SELECT is saved.
- The COMPILER PROGRAM must be rerun whenever SELECT is rerun or when a different group of 33 fault environments is to be simulated. It would not be rerun if different Saturn diagnostic programs were to be run with a given set of compiled faults.
- The FAILURE INJECTION PROGRAM is run only when a group of faults is to be compiled and simulated.
- The Executive program is run more frequently than the others, since this is the actual simulation. It is here that the user will specify the Saturn program to be executed, the initial conditions for the logic and the output options desired. It produces a tape which contains the binary simulation results and is the input to the Node-Edit program.
- The Node-Edit program is usually run once per simulation, but it may be used to edit different fault environments or to produce additional copies of simulator output.

INSTRUCTION FETCH TIME			ERROR MONITORS													INSTRUCTION	
PHASE	BIT TIME	CLOCK	EP01	EP02	EP03	EP04	EP11	EP12	EP13	EP14	EP15	EP16	EP18	EP19	EP20	SECTOR	ADDRESS
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	05	054
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	05	057
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	05	060
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	05	062
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	037
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	13	042
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	043
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	13	051
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	054
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	13	061
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	13	063
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	064
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	13	226
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	227
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	13	232
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	233
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	10	212
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	213
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	10	222
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	223
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	10	326
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	327
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	10	332
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	334
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	10	341
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	10	342
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	344
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	103
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	104
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	[illegible]
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	113
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	201
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	202
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	203
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	204
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	02	206
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	207
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	211
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	213
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	222
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	224
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	225
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	03	012

Figure 9. Typical Printout From Redundant Computer Simulation

- The two blocks in Figure 9 labeled "Evaluator Program" and "Evaluation Report" are proposed but not yet included in the system. The purpose of the Evaluator would be to determine the effectiveness of the operation code diagnostics, as measured by the diagnostic resolution, the hard-core requirements, and the time to diagnose. This evaluation would be accomplished by a programmed comparison of actual simulation results against user supplied data, in a manner yet to be defined.

Simulator Running Time. IBM 7090 Computer, 32K memory. (Assuming about 4,000 logic blocks are to be simulated):

- Select program - 5 minutes
- Fault Injection program - < 1 minute
- Compiler program - 6 minutes
- Executive program - 12 Saturn op codes per min. This includes complete simulation of each operation code for each of the 168 Saturn clocks per instruction cycle, so actually 2,016 Saturn cycles are simulated per minute.
- Edit program - running time depends upon the amount of simulation output. For a shor Saturn program (20 to 30 op codes) the Edit program could require about three minutes to edit all of the 33 fault environments plus the good machine.

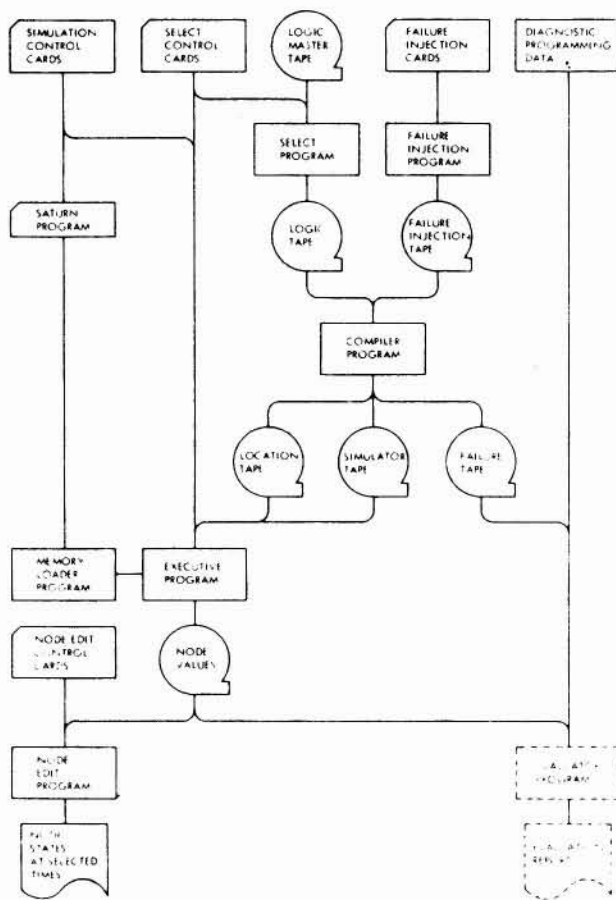

Figure 10. Saturn-V System Simulator Flow Diagram

PART II - SIMULATION

Introduction

Extensive use was made of the Saturn V System Simulator on the Saturn and related programs as an aid in evaluating computer operation in both normal and faulure modes. The simulator was used to verify the logical integrity of computer circuits and to evaluate proposed engineering changes. The computer self-test program and built-in test circuitry operation was examined with the aid of the simulator to determine the effectiveness of the computer error-detection system. Data identical to that available in normal checkout of a Saturn computer with its associated test equipment was generated by selected simulation runs and examined for diagnostic content, and additional data was compiled for use by the operator in diagnosing failure symptoms.

The Saturn V LVDC (Launch Vehicle Digital Computer) is a binary, fixed point, serial machine employing TMR (triple modular redundancy) to provide very high reliability for the Saturn V mission. The computer logic is organized in three identical parallel channels with almost 200 voting circuits located in such a way as to provide 2-out-of-3 voting on a modular level. As many as one third of the computer components theoretically could fail without causing a computational error in a TMR machine since the effects of each component failure in a channel are voted out by the equivalent components in the other two channels.

Since the effects of component failures tend to be masked in a TMR machine, built-in detection circuitry was instrumented in the Saturn V computer to detect disagreements in the logic status of the three channels and thereby determine the reliability status of the computer. These disagreement detectors were designed as three-input exclusive OR circuits and placed primarily across the three inputs of the voters.

An additional circuit checkout capability was instrumented by providing a simplex mode operation in which one channel is forced to a logical one level and a second channel forced to a logical zero level. Since the votes of these two forced channels cancel, the status of the third channel determines the logical status of the computer, and a component failure in this channel can be detected by normal checkout procedures used with simplex computers. Any of the three channels may be selected as the operating channel by a mode select switch.

During the subject studies, several hundred failures were simulated in the computer in both simplex and redundant modes. Although the results represent a relatively small statistical sample, they were sufficiently consistent to provide a high degree of confidence in the conclusions.

Simplex Simulation

In the Saturn V development program, the initial phase included the design, build and test of a simplex version of the computer for engineering evaluation. The purpose of this phase was to assure the logical integrity of the basic design and to determine by parametric measurements any marginal characteristics of the design.

Simulation of the Saturn V computer in its simplex mode of operation provided data similar to that obtained in the hardware tests with the simplex computer breadboard. The simplex simulation experiments fell into three primary areas: design verification, test program evaluation, and test point catalog generation.

Simulation provides certain advantages over hardware testing for preliminary engineering

evaluation of a new design. Some of the principle advantages are:

- Timeliness — Simulation data can be made available to the designer long before breadboards can be built and tested.

- Completeness — The speed and flexibility of simulation over hardware testing allows a more thorough examination of the circuit operation under variable operating conditions.

- Access — Unlike hardware testing, simulation provides complete access to all logical modes in a digital system for signal injection and status monitoring.

Design Verification. Portions of the Saturn V computer, such as registers and counters, were simulated individually to verify their logic integrity before the computer was simulated as a complete system. One of these preliminary simulations was a set of race-monitoring experiments in which a latch may be set or reset depending upon marginal timing characteristics of the input signals. A race condition is defined here as a (0, 0) on a complementary output nodes of latches or improper outputs on other sequential memory elements.

In one race-monitoring experiment, three race conditions were detected in a counter design. One was due to improper initialization conditions which were corrected in the computer initialization procedures. One was corrected by a design change. The third was a logical "don't care" condition which did not affect circuit performance but could result in a false disagreement detector error signal in the redundant mode of operation.

To verify the design integrity of the computer as a complete system, the computer logic was exercised in the simulator by a special test program (described in the next section) which was designed to assure that every logic component was exercised at least once during each cycle of the program. No improper conditions were discovered during this last phase of the design verification simulation program.

Test Program Evaluation. The simplex mode was designed to be a failure isolation mode of the Saturn V computer after detection of an error in the redundant (operational) mode. The failure isolation capability is based on channel-switching and module-switching techniques as well as on the analysis of data generated by the various tests. The computer test program was primarily designed to (1) detect logic failures in the simplex or redundant modes, and (2) generated sufficient data to allow a knowledgeable operator to diagnose the failure symptoms.

The basic organization of the test program was changed from the usual boot-strap functional exerciser to a component-oriented, sandwiched-subroutine format. The program was generated by failing components systematically (on paper) and deriving subroutines to check each and every failure. Instruction addresses were selected to exercise all drive lines in all storage sectors during the program cycle. The complete test program consisted of less than 500 instructions.

The resultant test program provides advantages over a functional program, although the work effort involved in generated it is considerably greater. The component orientation of the program requires fewer instructions. The distribution of the computer functions throughout the program, rather than lumping each function in a particular portion of the program, provides a better inherent capability for detecting intermittents.

During the subject study, several hundred selected and randomly chosen failures were simulated by a batch-simulation technique in which 33 failed machines were exercised simultaneously. Only single failures were injected into each machine. The simulator exercised the test program until the first failure detection, at which time the failure was flagged and simulation of the flagged machine discontinued. The simulator then continued exercising the remaining (unflagged) machines until the next failure occurred. This process was repeated until all failures had been flagged or (in the case of undetected failures) until the test program had been completed.

Of the injected failures, less than 10 percent were undetected by the test program. An examination of the undetected failures disclosed, however, that the majority of them would not cause a logic failure even in the simplex mode. Many of the undetected failures involved logic elements which were included in the computer instrumentation to insure against possible marginal conditions, especially in the area of timing. Others involved logic elements included to minimize power consumption, or which were simply redundant--residues of design changes that eliminated their functions but which were retained to minimize change costs.

After these types of failures were screened out, less than one percent of the simulated failures remained unaccounted for, resulting in a failure

detection efficiency of over 99 percent for the simplex computer/test program system. Changes in the test program were made to pick up most of these undetected failures.

A limited analysis of simulation results was performed to determine whether sufficient information was generated by the test to enable an operator to diagnose the error symptoms. Although it was concluded that diagnosis was feasible, the complexity of the failure isolation problem appears to preclude extensive precomputed correlation of symptoms and faults in the simplex mode.

Figure 11 is a curve derived from a typical simplex simulation. It illustrates the relationship between the percentage of detected failures injected into the simulator and the portion of test program completed. The high rate of failure detection during the early portion of the program can be attributed to (1) the necessarily high percentage of operation codes used early in the program, and (2) the existence of program-independent errors. The curve shows that the program could be considerably shortened if error detection was the only requirement. For example, the program could be truncated to the first 40 percent of the instructions while maintaining a 90-percent error detection capability. This characteristic, indicated by the dotted lines in the figure, is applicable to the construction of in-flight checkout programs. However, the full program is desirable from the standpoint of generating diagnostic data, and it provides better error-detecting efficiency.

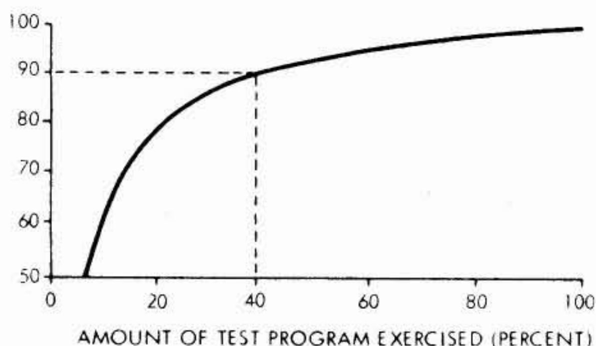

Figure 11. Typical Simplex Simulation Curve

Test Point Catalog Generation. The simulator was used to generate a test point catalog containing the logical values of 90 selected test points at each computer clock time while the simulator executed the 480 instruction test program. The completed catalog contained more than 15,000 logic values. The value of this catalog for laboratory checkout of the computer is that the state of the computer is predetermined at 90 nodes in its organization for a failure-free condition. Comparison of test data generated in laboratory checkout of the actual computers with the test point catalog has proven helpful in diagnosing symptoms.

Redundant Simulation

The redundant mode was designed to be the operational and failure-detection mode of the Saturn V computer. Again, as in the simplex-mode simulations, the computer logic was exercised by the test program to determine the effectiveness of the failure detection function. In its redundant mode, however, the computer provides error indications from the built-in disagreement detector circuits to aid in detecting and diagnosing error symptoms.

In addition to evaluating the effectiveness of the test program-disagreement detector system, the simulator was used to determine the optimum number and placement of disagreement detectors in the computer organization. Similar simulation experiments were performed to determine the optimum placement of voters in the TMR logic.

One of the major advantages of a TMR organization is its ability to perform in the presence of intermittent component and interconnection failures. If an intermittent failure occurs in one of the three TMR channels, its effects are not only masked by the voting function, but the computer recovers its initial (unfailed) state after the period of the intermittent has ended. An intermittent failure analysis was performed with the aid of the Saturn V simulator to determine the effectiveness of the computer detection system in detecting various classes of intermittent failures.

Detection System Evaluation. Although the redundant mode was designed to be the operational and failure-detection mode of the Saturn V computer, availability of a sufficient amount of failure data based on error-monitor indications appears to allow a high degree of failure isolation. Several hundred failures were simulated using the batch-simulation technique already described. Again, as in the simplex-mode simulation, only single failures were injected into each simulated machine. Unlike in the simplex simulation, however, each failed machine was exercised for the duration of the test program. As a result, a varying error-monitor pattern was generated for each failed machine.

Again, of the several hundred failures injected into the computer logic, less than 10 percent were undetected by the error monitors. As in the simplex simulation, these undetected failures involved either redundant logic elements or those which were included in the computer design to conserve power or insure against marginal conditions. A 99-percent failure-detection effectiveness was obtained after these types of failures were screened out.

The approach to failure isolation as examined in the redundant simulation is based on correlation of logic failures with error monitor patterns, pattern changes, and sequence of pattern changes. The simulation data indicated that about 75 percent of the failures could be isolated to a single logic module through examination of the error monitor patterns. About 90 percent could be isolated to one or two modules. In addition, examination of certain pattern characteristics--such as fixed-or-variable pattern, number of pattern changes during the test program, and sequence of error monitor changes as the test program exercises various portions of the computer logic provides an error resolution of one module for almost all of the simulated failures. In those cases where the test resolution exceeded one page, channel and/or module switching would provide the additional resolution desired.

Another important conclusion from the computer simulation is the apparent feasibility of constructing a diagnostic test program in which program branching is based on error monitor indications. The main program would be a short logic exercisor designed for efficient error detection only, and would operate periodically during the operational periods of the computer mission. If no error is detected, little operational time is consumed by the test. But if an error is detected, the program will branch to specific subroutines determined by the error monitor patterns.

Figure 12 is a portion of a typical print-out from the simulation of a redundant computer. The phase, bit, and clock time listed in the left-hand column is the instruction fetch time, but the simulator could be instructed to print out the actual time of occurrence of the error signal instead. The error monitor signals are represented by the 13 EP (error position) columns, and the instruction sector and address location by the right-hand columns.

The simulator was instructed to print out a new line every time an EP location changed state. Consequently, only a small portion of the test program is listed in Figure 12. The particular failures simulated in this run affected error monitor positions 12 and 19. Diagnostic information is contained not only in the generated EP signals but also in the instructions associated with a change of state of an error monitor and in the total number of changes in state, i.e., with the entire EP pattern.

Table I represents the results of another redundant simulation in which the simulated failures are associated with logic pages 1 and 2, and with error monitors 1, 2, and 3. An examination of the failure/monitor correlation alone (ignoring the additional diagnostic information given by the instructions and time sequences associated with state changes) indicates a high degree of resolution between the two pages. Error monitor combinations EP1 alone, EP1 EP2, and EP1 EP3 were associated with failures injected onto page 1, error monitor combinations EP2 alone and EP1 EP3 were associated with failures on page 2; and error monitor EP3 alone indicated a failure on either page 1 or page 2. Failures thus isolated to page 1 represented 39 percent of the simulated failures, those isolated to page 2 represented 36 percent, and those which could not be resolved between page 1 or page 2 represented 25 percent. However, the 25 percent of unresolved failures could then be resolved by an examination of the full pattern equivalent to that illustrated in Figure 12.

Many different types of symptoms were produced as a byproduct of the simulation experiments. All of these were analyzed to determine their individual and combined value in identifying logic signal failures. Table II gives a summary of these results. Signal failure identifications are based on the rearranged eight-diagnostic module configuration. Only unique signal identifications were tabulated.

Disagreement Detector Placement. The disagreement detection function was simulated by comparing the logic status of selected nodes in the channel containing the injected failures with corresponding nodes in a reference channel. Simulated removal or addition of disagreement detectors was accomplished by removal or addition of monitoring nodes from the selected node listing.

Just as packaging of computer componets on replaceable modules partitions the computer physically, placement of disagreement detectors within a machine organization partitions the computer diagnostically.

At present, no clearly defined ground rules exist which can be applied to optimally partition

INSTRUCTION FETCH TIME PHASE	BIT TIME	CLOCK	EP01	EP02	EP03	EP04	EP11	EP12	EP13	EP14	EP15	EP16	EP18	EP19	EP20	INSTRUCTION SECTOR	ADDRESS
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	05	054
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	05	057
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	05	060
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	05	062
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	037
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	13	042
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	043
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	13	051
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	054
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	13	061
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	13	063
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	064
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	13	226
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	227
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	13	232
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	13	233
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	10	212
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	213
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	10	222
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	223
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	10	326
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	327
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	10	332
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	334
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	10	341
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	10	342
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	10	344
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	103
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	104
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	111
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	113
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	201
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	202
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	203
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	204
A	09	Z	0	0	0	0	0	1	0	0	0	0	0	1	0	02	206
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	207
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	211
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	213
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	02	222
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	224
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	1	0	02	225
A	09	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	03	012

(ERROR MONITORS: EP01–EP20)

Figure 12. Typical Print-Out From Redundant Computer Simulation

Table I

RESULTS OF A TYPICAL REDUNDANT COMPUTER SIMULATION

	EP 1	EP 2	EP 3
EP 1	Page 1 22%		
EP 2	Page 1 5%	Page 2 23%	
EP 3	Page 1 12%	Page 2 13%	Pages 1 and 2 25%

electronic units into diagnostic modules. Logic simulation has been used, instead, to determine the characteristics of failed machines and the nature of error propagation in a digital system to provide data from which such ground rules might be derived. Two simulation experiments were performed during the study to trace failure propagation through the computer logic. Sixty-six simulated failures were injected into representative voter interfaces and error propagation was monitored by disagreement detectors placed at the input to every voter and at other selected logic nodes within the four modules of the AES computer. These nodes were selected on the basis of the total number of signal inputs to logic latches.

These experiments provided sufficient data to partition the computer into diagnostic modules although no change in the physical packaging of the computer modules was considered. The arithmetic module of the computer was partitioned further into three diagnostic modules, as was the control module. A comparison of error signal

propagation between four and eight diagnostic modules is shown in Table III for sample failures. Note that there is less likelihood of identical failure symptoms occurring for failures in each of the four physical modules if the additional diagnostic partitioning is instrumented. For example, a failure in physical module 2 and another in physical module 3 caused identical failure symptoms in physical modules 2 and 3 when the computer was partitioned diagnostically into four modules but no identical failure symptoms when the computer was partitioned into eight diagnostic modules.

Table II

SYMPTOM - FAILURE CORRELATION

No.	Observed Symptoms in Logic	Failures Identified (percent)
1	First Program Step of Detected Error	10.5
2	Final Error Pattern	26.3
3	Time of First Detected Failure	28.1
4	Final Error Pattern	20.2
5	First Three Program Steps of Detected Errors	63.2
	First Three Program Steps of Detected Error and Final Error Pattern	96.5
	First Program Step at Detected Error and Final Error Pattern	63.1
	First Program Step of Detected Error, Final Error Pattern, and Phase, Bit, Clock Time of First Detected Error	82.4

The extensive propagation of errors through the computer presented the greatest problem in isolating failures to a replaceable module. Propagated errors tend to be sensed by many detectors even though these detectors are not directly associated with the logic containing the failure, thus masking the source of error by "overdetection". An approach suggested during the course of the study of clocking the detectors only at the time that the associated logic is being used was found to require too much additional timing circuitry to be practical. Bit gates, phase gates, and in some cases even program step identification were found to be required to accomplish the desired detector timing.

Table III

ERROR SIGNAL PROPAGATION

Functional Partitioning				
Interface Failure In Module	Symptoms Will Occur In Functional Modules			
Timing	1	2	3	4
1		2	3	4
2	1	2	3	
3	1	2	3	
4	1	2		4

Diagnostic Partitioning								
Interface Failure In Module	Symptoms Will Occur In Dynamic Modules							
Timing	1	2	3	4	5	6	7	8
1				4	5	6		8
2		2	3	4	5	6	7	8
3		2	3	4	5		7	8
4		2		4	5			8

The Saturn-V disagreement detectors are clocked every like clock time (for example, any one disagreement detector may be clocked every x-time, another every y-time, etc.). As a result, detectors are sensing for disagreements between the simplex modules of TMR trios even at times when those modules are not being used by the program.

Error propagation has also been the major problem in attempting an optimum placement of disagreement detectors. Although failure isolation to a replaceable module level has been found to be feasible in the computer by reorganization on a functional basis and by redesign of the Saturn-V disagreement detectors, means must be

found to prevent the error from propagating from one module to another and thereby destroying the isolation (as in the case of timing signals). An approach was investigated in which each of the logic modules was partitioned into two or more diagnostic sections by placing additional detectors internal to the module to provide required isolation information.

The logic simulator was revised to allow flexible diagnostic partitioning and used to provide data for optimum placement of disagreement detectors.

A logic simulation was designed to determine the optimum placement of disagreement detectors in the TMR logic. A total of 32 voters were failed and the failure data analyzed to determine the logic level to which the failures can be localized. The specific voters to be analyzed were chosen as representative of the various types of combinational and sequential circuits which would be "inputted" by the voted signals. The instruction and computer time when any of the module interface disagreement detectors sensed a failure was tabulated. An analysis of the simulation results showed that:

- Fifty-three percent of the voter failures could be identified by knowing which disagreement detectors had sensed the failed conditions.

- Forty and seven tenths percent of the voter failures could be identified by knowing the program instruction and computer time of first detection in addition to which detectors had sensed the failed conditions.

- Six and three tenths percent could not be identified.

The partitioning of the reorganized computer resulted in using approximately 120 voters at the module interfaces. The simulation described assumed disagreement detectors at the input of each voter and nowhere else. The 6.3 percent of the voter failures which could not be identified was due to error propagation within a module and signal feedbacks between modules, resulting in identical error patterns for different failures.

This problem was alleviated by placement of additional disagreement detectors within the modules and at the module interfaces. To determine the number and location of the intramodule detectors, the four computer modules of the reorganized computer were divided into equivalent diagnosable subunits by physical count of the signal inputs to each of the latches and tratches in each of the modules. Table IV summarizes the results of this count and indicates a measure of the unbalance of signals and voters (disagreement detectors) in each module.

Table IV

SIGNALS, LOGIC, AND VOTERS

Module		Signal Inputs	Latches, Tratches	Voters (DD'S)
Number	Name			
1	Memory and Read	459	27	17
2	Arithmetic	1213	74	9
3	Control Timing	387	34	26
4	Operation and Decoder	720	69	45
	Timing (Distributed among four Modules)	135	13	23
	Total	2914	217	120

Of particular interest is the ratio of the total number of signal inputs to the total number of voters (or disagreement detectors since the DD's were located at the voter inputs). This ratio was found to be 24:1. Using this figure as the basis for organization of equivalent diagnosable subunits, approximately 21 additional disagreement detectors were required. Their distribution and effect on the detector-to-signal ratio is shown in Table V. The ratios are average values, which may be misleading because the additional detectors were chosen on the basis of individual circuit sizes within the module and on the basis of use and criticality. The effect of these additional 21 disagreement detectors was determined by simulation.

Table V

ADDITIONAL DISAGREEMENT DETECTORS

Module		Basic Ratio	Added DD's	Modified Ratio
Number	Name			
1	Memory and Read	27.0	1	25.5
2	Arithmetic	134.8	16	48.5
3	Control Timing	14.9	2	13.8
4	Operation and Decoding	16.0	2	15.3

Based on component packaging density and intermodule wiring considerations, the Saturn V computer was repartitioned into four modules. Approximately 105 disagreement detector trios have been defined for intermodule failure detection. Table VI shows the distribution of these detectors in the four modules.

Table VI

DISTRIBUTION OF DETECTORS

Module	Function	DD's
1	Memory and Memory Interface	39
2	Arithmetic	9
3	Address Registers	27
4	Control	30

Voter Placement. The voting function was simulated in the Saturn V simulator by forcing the logic status of selected nodes in the channel containing the injected failures to agree with corresponding nodes in a reference channel. Error propagation is thus allowed only from the point of failure injection to the first "voter" node in the data flow.

Optimum placement of voters for restricting error propagation (a reliability factor) or for diagnostic capabilities can therefore be investigated with the aid of the system simulator. This is an area which was not thoroughly examined in the simulation studies.

Intermittent Failure Analysis. Several experiments were made to determine the sensitivity of the computer logic to intermittent faults. These intermittents were made to vary in duration from 500 nanoseconds (one clock time) to 5 milliseconds. These intermittent faults were specified at randomly chosen points of combinational and sequential logic circuits in the arithmetic-instruction and multiply-divide units. A total of 535,798 intermittent failures were simulated in the logic to give a realistic statistical sample.

For each intermittent, a record was kept of the time of each detection, the number of detections and failures which caused a difference from a "good" machine. From these records, the probability of detection was calculated. Table VII summarizes the results of the simulation.

Table VII

DETECTION OF INTERMITTENTS

Logic Simulated	Failures Causing Difference From "Good" Machines	Percent Of Total Failures	Number Of Failures Detected	Percent Detected Of Total Failed
Mult-Div	22,376	8.3	1,122	5.0
Instr. Ctr.	44,704	17.5	252	0.5

This tabulation shows that many intermittent failures will have very little or no effect on the correct operation of the logic circuits, i.e., 8.3 percent of the total failures injected would cause the computer to perform incorrectly. An analysis showed that this "masking" of failures is primarily due to the large use of combinational logic and the method of clocking the "AND" gates which feed the sequential circuits. This table also shows the large difference in error detection sensitivities between the two modules, and suggest a need for a more efficient partitioning in the modules.

Data derived from the simulation runs was used to calculate the detection efficiencies of the disagreement detectors. Figures 13 and 14 give this plot of detection probabilities versus failure duration. The figures in Table VII are heavily weighted by a large number of short failures,

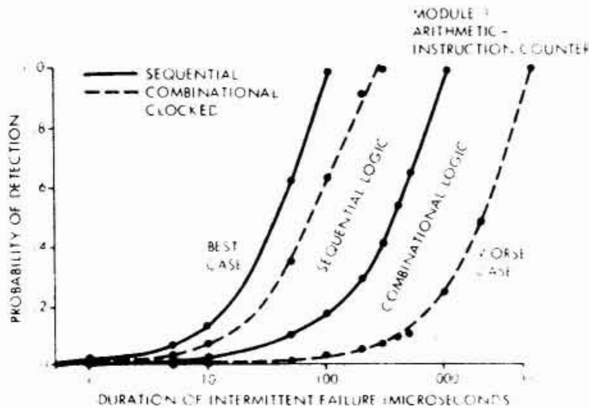

Figure 13. Detection Efficiency (By Computer Disagreement Detectors)

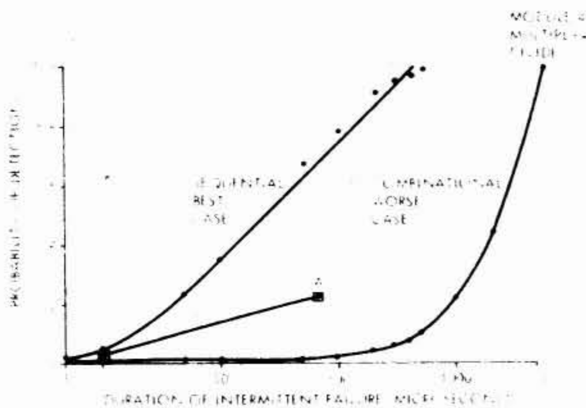

Figure 14. Detection Efficiency (By Computer Disagreement Detectors)

leading to a low average detection probability. Figures 13 and 14 indicate these probabilities for various lengths of intermittents. The "best" and "worst" case detections are shown to give the spread in detection efficiencies.

A summary of the results obtained in simulating intermittent failures is given below:

- There is a smaller probability of detecting intermittent failures in combinational (AND) type circuits than in sequential (latch) circuits.

- There is a very small chance of detecting a single occurrence intermittent failure on a logic page. One experiment verified this finding by detecting six out of a total of 33 randomly selected failures which had intermittents lasting for one computer word time. The detection probability is low because of the fact that many intermittents have no effect on the logic (they do not make the "failed" machine different from a "good" machine) and further, because detection is program dependent, in that the logic must be exercised by appropriate instruction and data for the failure to be indicated by a disagreement detector.

- There is a wide variation of error detection sensitivities between modules. This sensitivity could be equalized to provide a more efficient error detection organization.

- For the fault locations chosen and the programs executed, the faults caused block outputs to differ from those of the unfailed machine only about 10-20 percent of the time.

- For these locations and programs a fault existing for 0.5 microsecond (one clock time) was virtually undetectable; one existing for 0.5 to 1 millisecond was about 50 percent likely to be detected; one existing for 4 to 5 milliseconds was almost 100 percent likely to be detected.

- The disagreement detectors are clocked, thus having a 25 percent duty cycle. Changing the simulation model to give a 100 percent duty cycle made no significant improvement in failure-detection ability in these experiments.

- During analysis of the results of the multiply-divide exercise program used for these runs, it was discovered that the machine would fail to recognize a MPY or DIV instruction given during the instruction cycle in which the results of a prior MPY or DIV would normally be stored; for example, by the sequence CLA, MPY, CLA, SHF, ADD, MPY. Such an instruction sequence would not normally be programmed, but could occur in a diagnostic exercise program like this one. When the hardware machine was found to agree with the simulation, analysis showed the timing relation which caused the action.

Acknowledgements

R. E. Forbes, C. B. Stieglitz and D. H. Rutherford for their technical guidance in the design of the Saturn V. Fault Simulator. L. H. Tung

for the formulation of certain basic simulation techniques. R. E. Ide for a major portion of the Fault Simulator programs and documentation. M. Ball and R. M. Lewis for the design of the simulation experiments and for the evaluation of the simulation data.

References

R. E. Ide, R. J. Suhocki, "User's Manual, Logic System Simulator", IBM No. 65-578-01, January 8, 1965.

IBM Dept. 578, "Final Report Work Package 3860", IBM No. 65-578-03, January 12, 1965.

P. W. Case, H. H. Graff, L. E. Griffith, A. R. Leclercq, W. B. Murley, T. M. Spence, "Solid Logic Design Automation", IBM Journal, Volume 8, No. 2, April 1964, pages 127-140.

"AES-EPO Study Program-Final Study Report", IBM No. 65-562-012.