NAG 1-613

Error Propagation in a Digital Avionic Processor

A Simulation-Based Study

D. Lomelino and R.K. Iyer

Computer Systems Group
Coordinated Science Laboratory
1101 W. Springfield Avenue
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801 U.S.A.
(217) 333-9732

ARPANET: iyer%uicsg%uiuc.arpa
CSNET: iyer%uicsg%uiuc.csnet
UUCP: uiucdcs!uicsg!iyer

Abstract

This paper describes an experimental analysis to study error propagation from the gate to the chip level. The target system is the CPU in the Bendix BDX-930, an avionic miniprocessor. Error activity data for the study was collected via a gate-level simulation. A family of distributions to characterize the error propagation, both within the chip and at the pins, was then generated. Based on these distributions, measures of error propagation and severity were defined. The analysis quantifies the dependency of the measured error propagation on the location of the fault and the type of instruction/microinstruction executed.

Keywords: Simulation, fault-injection, error propagation, distributions, pin-level fault models.

1. INTRODUCTION

The problem of testing and validating real-time digital flight control systems remains quite intractable at this stage. A major reason is that neither the error generation process nor the fault propagation problem is well understood. The mechanisms involved are highly complex and hence not easily amenable to analytical modeling. An experimental study can provide valuable insight and help develop a structured basis for analytical modeling.

This paper describes an experimental analysis to study error propagation from the gate to the chip level. The target system is the Bendix BDX-930, a digital avionic miniprocessor. The processor is simulated using an event-driven, gate-level logic simulator developed at NASA-Langley. The BDX-930 is used in a number of flight control avionic systems, notably SIFT [Wensley78] and AFTI F-16 [McGough81]. Fault tolerance is achieved by replication of the processing and voting in software. In this study, only the CPU of the BDX-930 is used.

The general objective of this study was to develop a systematic experimental methodology to quantify error propagation from the gate to the pin level. Particularly, we wished to do the following:

(1) Differentiate between faults in various functional units.

(2) Determine the relationship between error propagation and instruction and microcode execution.

(3) Compare error propagation at the internal gates to that at the output pins.

(4) Determine the validity of a single fault model at the output pins.

An approach to gather and model error propagation information is described. The approach can, in principle, be adapted to other systems. Error activity data for the study was collected by comparing the non-faulted (master) simulation run to each individual faulty simulation run. Data was gathered for faults distributed throughout one of the bit slice processors. A family of

distributions to characterize the error propagation, both within the chip and at the pins, was then generated. Based on these distributions, measures of error propagation and severity were defined. The analysis quantifies the dependency of the measured error propagation on the location of the fault. We also show the nature and extent of the dependency of error propagation upon the type of microinstruction executed, the assembly level instruction executed, and the fault-free gate activity.

1.1. Related Research

A series of experiments aimed at error analysis through fault insertion have been conducted by several investigators at the NASA AIRLAB test bed facility. A summary of these experiments is given in [Shin84]. [McGough81,83] and [Lala83] study the evaluation and modeling of fault latency in digital avionics systems. These studies were aimed at determining the degree of fault latency in a redundant flight control system. In [Shin84] a new experiment to study fault latency distributions through hardware fault injections is described. Experience gathered from these studies shows that the data generated can provide considerable insight into error manifestation. Another interesting study is [Courtois79] which describes a simulation experiment to determine the efficiency of a number of error-detection mechanisms.

An important question not addressed in the above experiments is how to quantify error propagation from the gate to the pins? Apart from furthering the knowledge of error propagation in the CPU, this information is crucial for developing effective pin-level fault models for use in validation and testing of flight critical digital systems. Currently, single pin-level fault injection is used in FTMP. In addition, this information may aid in developing CPU test stategies as well.

1.2. Following Sections

Sections 2 and 3 contain a detailed description of the experimental procedure and measurements. Sections 4 and 5 describe the measures by which error propagation is characterized. Section 6 quantifies the effect of the physical location into which the fault is injected (i.e. the logic unit), on error propagation. Section 7 describes the analysis of error propagation according to which instruction and microinstruction is executed. Section 8 compares the error activity at the output pins to that for all the gates within the chip. The final section summarizes the important results and draws conclusions.

2. THE EXPERIMENT

2.1. Synopsis

The target system chosen for studying error propagation is the BDX-930 digital avionic mini processor. Within this system, a single chip was selected for fault injection and error propagation data collection (AMD 2901 bit slice processor [AMD81]). As it is physically impossible to inject faults into the actual gates within the chip, simulation was chosen as the avenue for data collection. An event-driven, gate level, unit delay logic simulator developed at NASA-Langley was employed [Migneault85]. A permanent single stuck-at fault model was employed for the purpose of fault injection, though the methodology is expected to be applicable to other fault models. Faults were injected into each of the logic units of the AMD 2901 i.e., the RAM shift, the Q shift, the multiplexer (MUX), the arithmetic logic unit (ALU), the carry-propagate unit, the output data select unit, the destination control, the ALU control, and the source control. Such fault injection not only permitted the study of error propagation in a general sense, but also helped to determine the effect of fault placement on error propagation.

Error activity data was collected by first simulating a fault-free circuit, next simulating the circuit again with a single injected fault, and finally by comparing the simulation output for differences. These measured differences permit error propagation to be quantified. The simulation was conducted for 50 clock cycles. This period of time was found to be satisfactory for determining stable distributions and values for the measured quantities.[1]

The following subsections describe in detail the circuit, the simulator, the fault model, fault placement, and data collection. A detailed description of the circuit and error detection is given in Appendix A.

2.2. Simulation

The devices[2] available in the simulator for circuit descriptions include: simple gates (i.e., AND, NAND, OR, NOR, XOR, XNOR, and NOT), tri-state devices, and flip flops. Each device is allowed a unit delay. This is a reasonable assumption for the simple gate and tri-state devices, though flip flops would normally have longer delays (maybe up to an order of magnitude longer). This inaccuracy proved relatively unimportant since (as described in sec. 2.3) no flip flops were selected for fault injection, and error propagation through the RAM (e.g., due to a fault in the RAM shift), was found to be minimal in the measured period. Such propagation would be important in studying the behavior of latent errors. Specific experiments toward this end are reported in [McGough8183] and [Lala83].

Since the simulator is event-driven, the output is a list of devices that changed state (and their new output values) rather than the complete list of devices and their values for the time slice within the clock cycle under examination. This type of simulation minimizes both computation time and the volume of output data. Each event in this list contains a device name, the time

[1]A detailed sensitivity analysis in the range 5 to 2000 clock cycles was conducted to ensure that the results were not strongly influenced by the choice of this time.

[2]Note that the term 'device' here means gates, flip flops and the like and not a transistor.

slice in which it changed state, and the new logic value it now has. These events, corresponding to a change of state, will be referred to as device *firings*. An erroneous change of state, therefore, is referred to as a device *misfiring*.

The single stuck-at fault model was employed in this experiment. This model is well studied and has so far proven to be an acceptable representation of real physical faults, especially for systems like the BDX-930 which use no CMOS or VLSI technology. Furthermore, this is the only fault model that this version of the simulator is capable of, though later versions are capable of simulating other fault models. More detailed experiments with other, more complex fault injection schemes are planned for subsequent investigations.

2.3. Fault Injection

Next the problem of where to place a fault for an individual simulation run was addressed. In general, one would use a random selection process to inject the fault, but due to the number of simulated fault runs collected and the relatively small number of devices available for fault selection, faults were injected through most of these available devices.

An explanation is required here. There are 466 devices used to describe the AMD 2901 chip for simulation. More than half of these devices are used to describe the RAM. Since fault latency becomes a concern when measuring error propagation through a memory subsystem, experiments were conducted to determine whether or not error propagation would be measurable in the time frame of the desired sample length. These experiments showed that errors were not propagated through the RAM or Q register. Therefore, it was decided to leave these logic units fault-free. Doing this reduced the number of devices available for fault injection to less than 200.

Due to difficulties in simulating the connection of tri-state device outputs, it was necessary to insert some bus gates that do not correspond to any real gates in the AMD 2901 circuit. In order to avoid selecting any of these pseudo gates, the list of gates for fault injection was taken

directly from Bendix's circuit diagram. Using this method, a list of 150 gates was compiled for fault injection. Since 150 is a manageable number, faulty simulation runs were done for all 150 gates.

2.4. Error Detection

Error propagation is detected by comparing a non-faulted simulation with a faulty run. Results of the comparison are placed in a difference file. Details of the error detection are given in Appendix A.2. Briefly, three possible types of error propagation are detected through simulation:

(1) There is a gate firing in the master (non-faulted) simulation run but not in the faulty run,

(2) There is a gate firing in the faulty run that did not fire in the master run, and

(3) There is a gate firing in both runs for the same time slice but the gate takes on different logic values.

One should note that in some cases the difference file will be empty. The empty file corresponds to the fault remaining undetected, or to the fault being a latent fault, at least for the time period simulated. For the initial run of 150 individual faults, 78.7% produced error propagation detected within the chip, and 66.7% produced errors that propagated to the output pins. The simulated period corresponds to approximately 1% of the test program. For the whole self-test program, the McGough study found a 92.0% gate-level coverage and 97.6% component-level coverage [McGough83b]. Clearly, faults tend to produce errors quickly once inserted into an active system.

3. MEASUREMENTS

A total of 150 simulation runs (one for each faulted gate) were performed for the purpose of studying error propagation. Each simulation (duration 50 clock cycles) contained a single

stuck-at fault somewhere within the circuit. Altogether, 4 sets of simulation experiments were conducted, consisting of 150 simulations per set. Each set of 150 simulations is referred to as a *sample* (i.e. the overall experiment consists of 4 samples). The input to the simulator was a system self-test program developed at Bendix and modified at NASA-Langley.

Each of the four samples uses a different starting point within the self-test program. The dependency upon program locality was minimized by varying the program starting points for simulation. The four starting points correspond to four individual subtests within the self-test program. These subtests include:

the cyclic RAM test,
the CPU test,
the ALU test, and
the memory address processor test.

Starting points were chosen at the beginning of these subtests in order to maintain a close tie with reality. If the program was entered at some random point, the integrity of the resulting data would be questionable. Since there was not a significant difference between the samples, only the results of the CPU test are presented.

In order to determine the effect of the location of the fault upon error propagation, these 150 faults were distributed among nine of the logic units of the AMD 2901. The logic units faulted include: the RAM shift, the Q shift, the multiplexer (MUX), the arithmetic-logic unit (ALU), the carry-propagate unit, the output data select logic, the destination control logic, the ALU control logic, and the source control logic. The number of faults injected into each logic unit varied between 4 (for the source control logic) and 46 (for the ALU).

Information concerning the instruction and microcode activity was collected concurrently with the error activity data. This data included the number of gates firing in the non-faulted case (referred to as *non-faulted gate activity*), the time slice in which the activity occurred, and the microaddress accessed. Knowing which microaddress was accessed uniquely helps one to identify

the executed microinstruction, which in turn allows one to determine the type of microinstruction executed. Finally, by examining the sequence of microinstructions, one can determine which macro (or assembly) level instruction was executed.

In summary, the information on error acticity is merged with the instruction/microinstruction activity information described above, for each of the samples collected. Each data record used to characterize error propagation contains the following information:

the clock cycle (between 1 - 50),
the time slice (between 1 - 3500, there are 70 time slices/clock cycle)
the number of gates firing in the non-faulted case for each time slice,
the number of misfiring gates for each time slice,
the microaddress accessed,
the type of microinstruction executed, and
the assembly level instruction executed.

Note that this data is collected for each fault injected into the system. All 150 simulations may be treated as a group to obtain the error propagation measurements for an average fault, or split into subgroups for analysis by logic unit or the instruction/microinstruction executed. The analysis of the collected data is discussed in the subsequent sections.

4. PRELIMINARY ANALYSIS

Preliminary analyses were performed upon the error propagation data collected to determine the pattern of error propagation versus time. Once a general understanding of the error propagation patterns was achieved, a method of quantification was developed.

4.1. Error Propagation versus Clock Cycle

A first level analysis of error propagation may be done by plotting the error activity versus time (measured in clock cycles). Figure 1a shows a plot for the non-faulted gate activity for comparison with the error activity plot presented in Figure 1b. One should immediately notice the similarity in shape between these two plots. Note the consistent matching of non-faulted gate

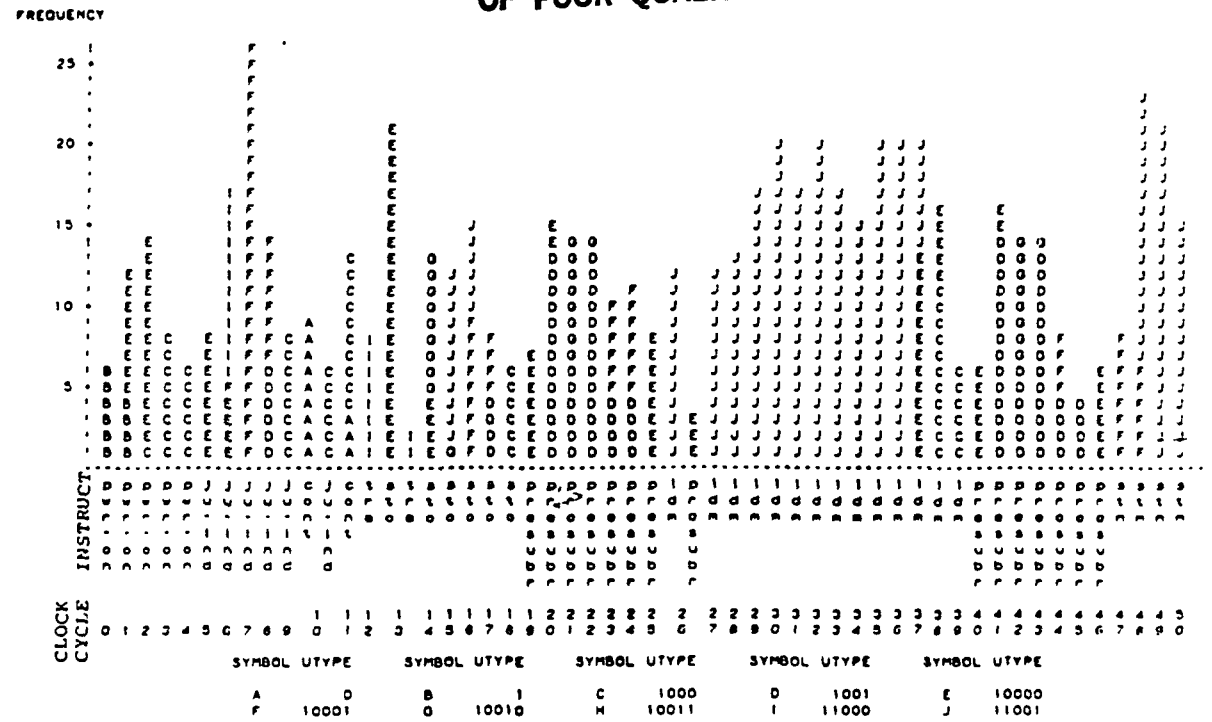
activity peaks with error activity peaks, and of non-faulted gate activity valleys with error activity valleys. Close correlation between non-faulted gate activity and error activity is indicated. Also shown in these figures are the instruction and type of microinstruction being executed for each clock cycle. The instruction is labeled across the horizontal axis, as is the clock cycle. The type of microinstruction may be read from within the bar-graphs themselves. Each letter that makes up the bars represents one type of microinstruction. The table relating the letters to the corresponding microinstruction type may be found below the graph. Note that sometimes there are two letters found in the same clock cycle. They are due to a slight misalignment of the clock cycle information and the microcode information due to the staistical analysis package [SAS82].

As for the cause of the oscillatory behavior of the non-faulted gate activity, it can be explained by examining the microinstructions that are executed within the specified instruction. In general, the peak gate activity occurs during an instruction prefetch or some other memory access. The reason appears to be the concurrent activity in the processor. It is quite reasonable that parallel processes like concurrent memory access and register transfer should cause more gate activity, which in turn causes more error activity under the influence of an injected fault.

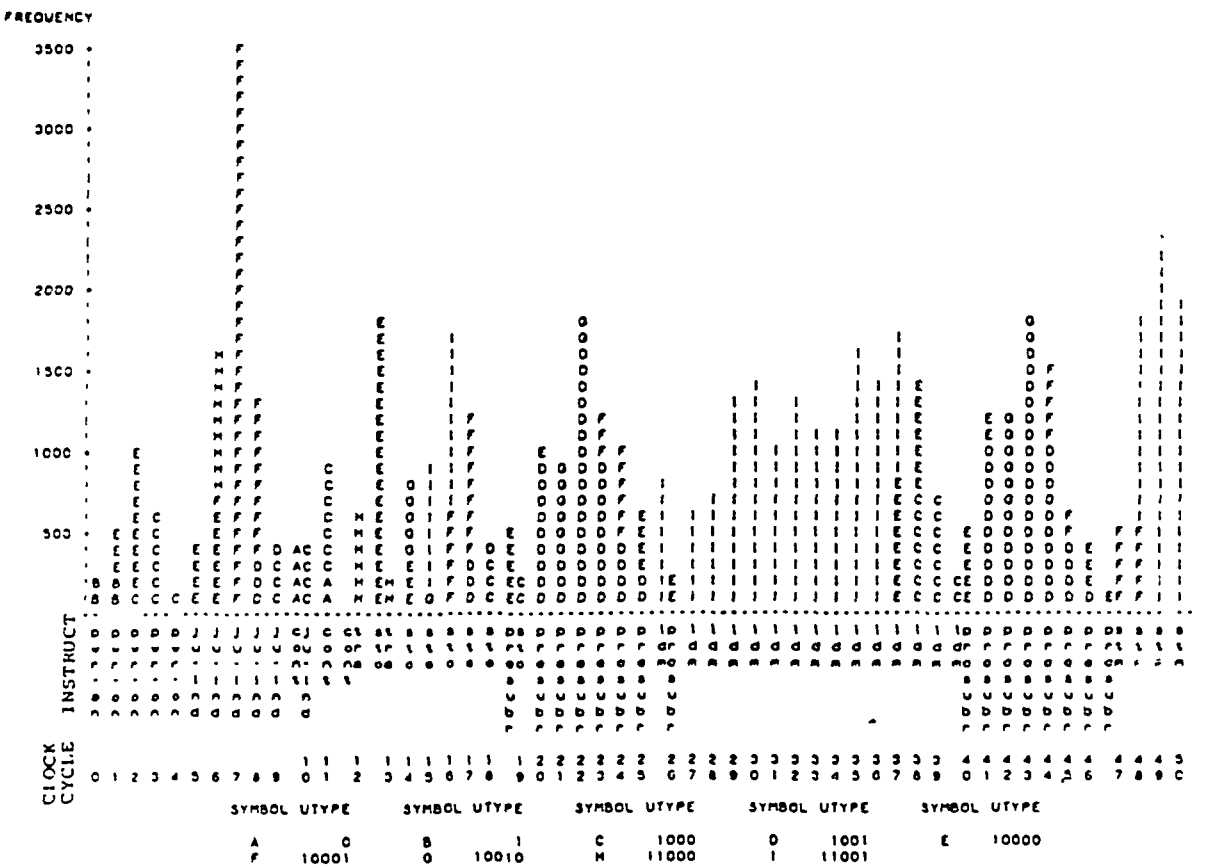
4.2. Error Propagation versus Time Step

Figures 2a and 2b characterize the speed of error propagation within a clock cycle. Given that a misfire will occur in a certain clock cycle (i.e., the fault does not remain latent), the probability of a misfire can be determined by reading the value of the cumulative percent column from the appropriate plot. For example, the probability of a fault resulting in an erroneous gate value in 10 time steps is 0.37.

These plots were generated by overlaying the 50 clock cycles in the sample and plotting the frequency of misfires for each of the 70 time steps. Most often there was no activity beyond 30 or so time steps/clock cycle. Tests showed that there was not significant variability between one



(a) For non-faulted case



(b) For all 150 faults

Figure 1: Error propagation by clock cycle

clock cycle and another for the measured interval. The misfire activity was found to be close to a Gamma distribution. This appears to be due to the inherent delay between a normal gate firing and its corresponding misfirings, should it be faulted.

5. QUANTIFICATION OF ERROR PROPAGATION

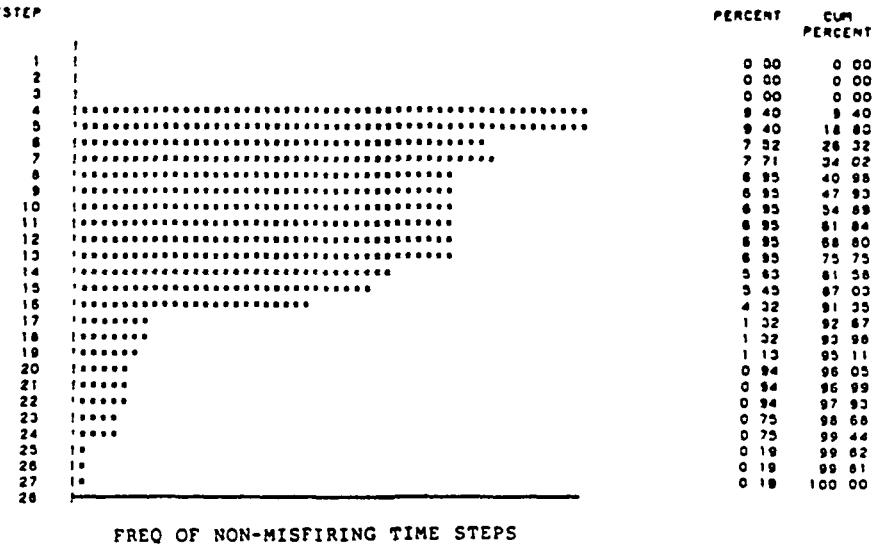
In this and the following sections, error propagation is quantified by determining useful parameters by which to measure it in a standard manner. Having been defined, these parameters are used to examine error propagation first for all the gates internal to the AMD 2901, and then for the output gates (to determine the accuracy of the pin level fault model). Through measuring error propagation for the internal gates, characteristic distributions can be developed for an "average" fault i.e., the average over all the 150 faults.

In general, two measures are necessary to quantify error propagation. Assuming there is a fault in the circuit, the probability of an error occurring should be quantified. Further, one also needs to know the severity of the error. This result is presented for different fault locations, and for different instruction/micro-instruction activities. Note that the mere existence of a fault in the system does not necessarily imply that there will be error activity, as some faults remain undetected throughout the entire sample.

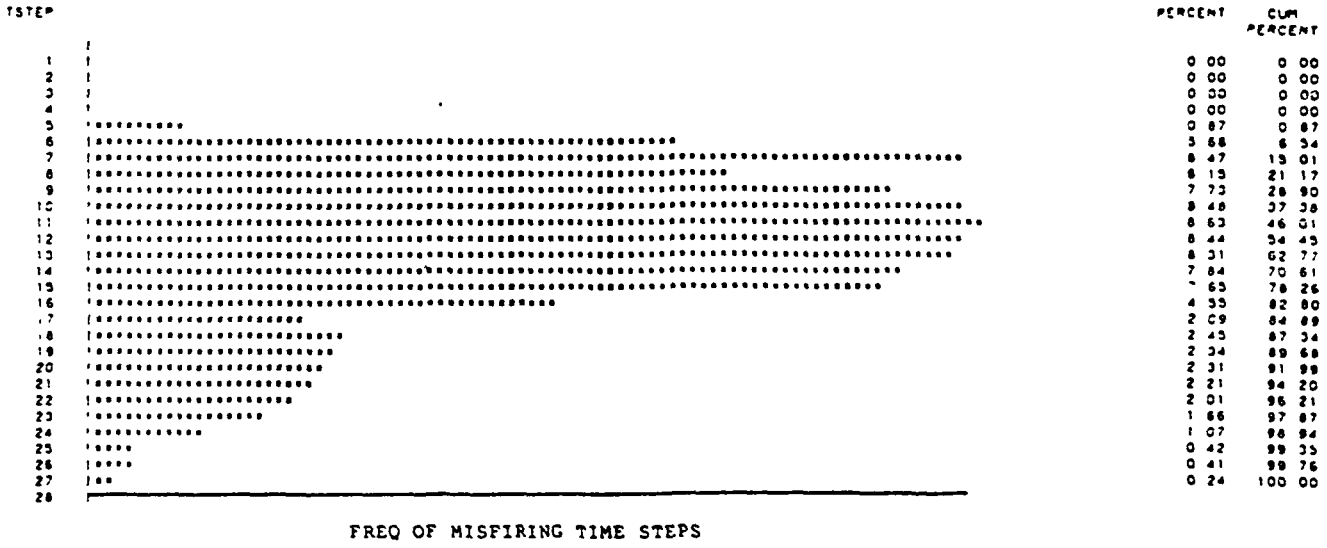
The following measured parameters were used to determine error proagation:

F	Existence of a fault in the circuit.
G	Non-faulted gate activity.
g	The number of non-faulted gate firings in a time step.
M	Misfire (error) activity.
$\overline{M}$	No misfire activity.
m	The number of misfiring gates in a time step.
S	The severity.

The G, M and $\overline{M}$ are binary variables while g, m and S are numeric variables.



(a) For non-faulted case



(b) For all 150 faults

Figure 2: Error propagation by time step

The non-faulted gate activity was an important reference for quantifying error propagation. Given a fault, the following measures of error propagation were defined:

(1) The first measure was the distribution of the number of time steps with at least one misfire. The distribution is plotted as a function of non-faulted gate activity. Thus we obtain the probability that, in a time slice, there is a misfire and non-faulted gate activity is equal to g. This measure is denoted by

$$M_g = Pr[(G = g) \text{ and } M] \mid Misfire.$$

Note that only misfiring time steps are counted above. An example distribution over all 150 faults may be found in Figure 3a.

(2) The remaining time steps (i.e. those not counted above) contain no error activity in the presence of a fault. The distribution of this data gives the probability that a time step has a certain non-faulted gate activity and no concurrent error activity (i.e. no misfires). The notation for this measure is:

$$\overline{M}_g = Pr[(G = g) \text{ and } \overline{M}] \mid NoMisfire.$$

An example distribution may be found in Figure 3b.

(3) From the information contained in Figures 3a and 3b, the probability of a misfire (given there is a fault in the circuit) can be computed as follows:

$$\frac{M_g}{M_g + \overline{M}_g} = Pr(M \text{ and } G = g).$$

An example distribution for the CPU subtest within the self-test program is shown in Figure 3c at the end of this section. Note that the probability of a misfire is still a function of the non-faulted gate activity.

(4) Finally, the severity of error activity is computed as follows:

$$S(m) = \frac{m}{m + g}.$$

This severity measure is computed for each time step. The mean severity is then computed

across the whole sample, or portion thereof (e.g. for all time steps having a certain system activity such as the power-on instruction or a register transfer microinstruction type). An example plot for the severity measure is shown in Figure 3d.

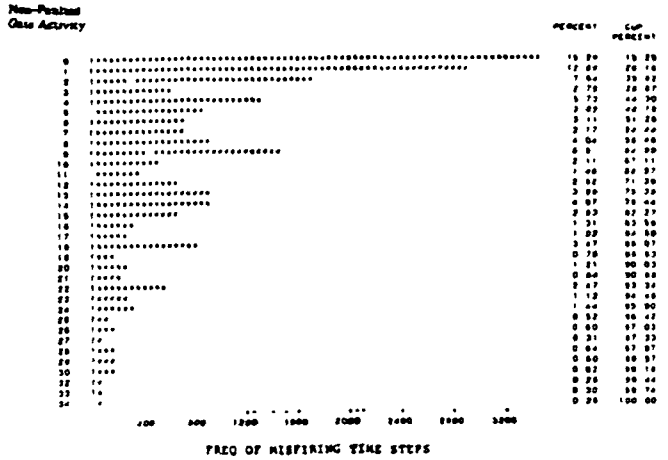
Having obtained the distributions for the probability of a misfire and the severity thereof, one can clearly see the relationship between the error activity and the non-faulted gate activity. The error activity is found to be strongly correlated to the non-faulted gate activity for the specified time step. The severity of a misfire has an inverse correlation.

6. ANALYSIS: EFFECT OF FAULT PLACEMENT

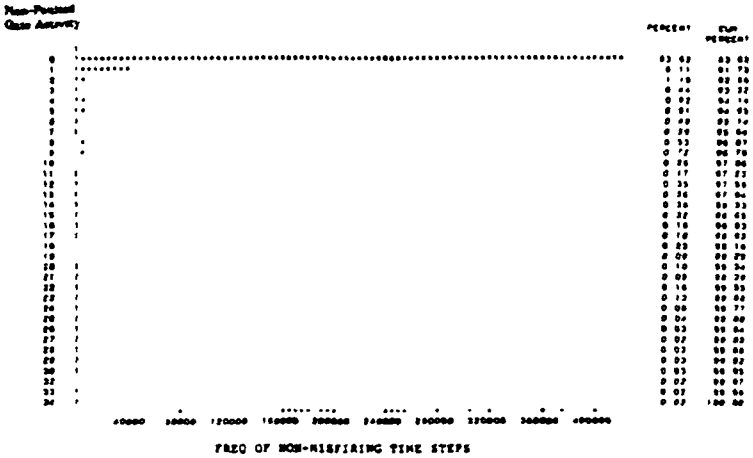
An intuitive determinant of error propagation activity is the location in which the fault occurs. For the analysis of the effect of fault placement upon error activity, two representative logic units are selected, and these are compared with the distributions and measures for the "overall" distribution (i.e. Figure 3).

The units selected are the ALU and the ALU control. As for the other logic units, the MUX errors behave very similarly to those in the ALU. This behavior is not surprising, as the MUX outputs feed directly into the ALU. The destination control unit and source control unit errors behave quite similarly to those for the ALU control. Finally, the RAM shift, the Q shift, and the carry-propagate units all have minimal error propagation due to their respective geographic locations. The errors due to the RAM shift and the Q shift tend to pass into the RAM or Q register respectively, and are not propagated further within the time frame of the sample. Due to the closeness of the carry-propagate faults to the output pins, there is little error propagation possible (since there is no feedback, as is the case for the output select unit). A summary of the percentiles for the measures for various logic units may be found in Table 1.

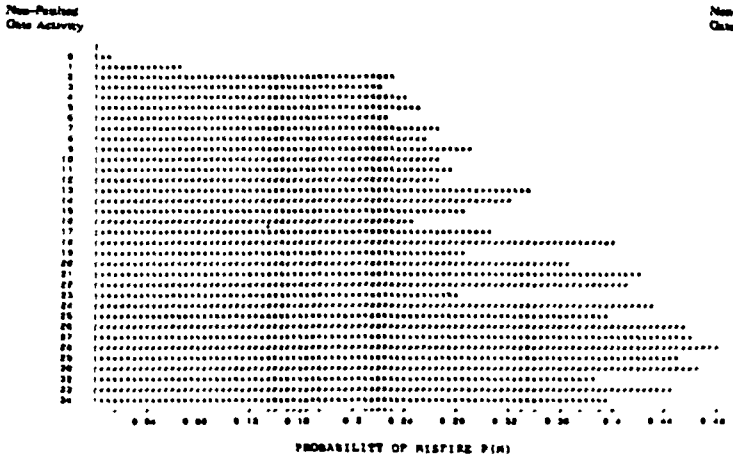
Close examination of example misfire probability distributions, found in Figures 4a and 4b, indicates that faults in the control unit are more likely to cause a misfire. This is a reasonable



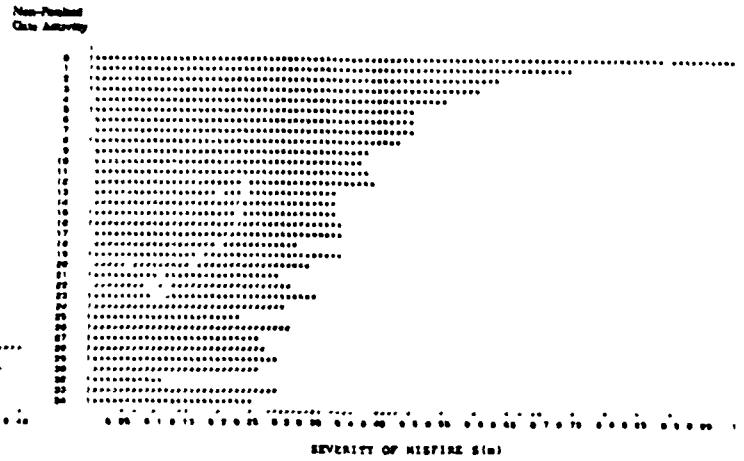
(a) Misfire frequencies for all 150 faults



(b) No-misfire frequencies for all 150 faults



(c) Misfire probability for all 150 faults



(d) Misfire severity for all 150 faults

Figure 3: Measured error propagation and severity

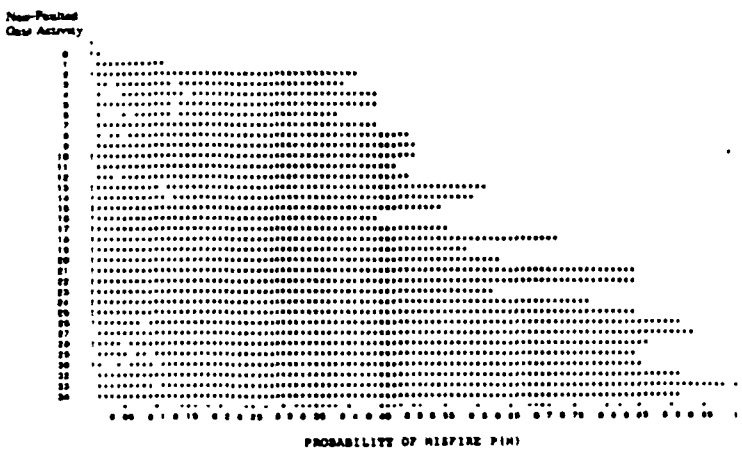
LOGIC UNIT	PROBABILITY P(M)			SEVERITY S(m)		
	50%	75%	95%	50%	75%	95%
RAM shift	0.04	0.06	0.18	0.38	0.10	0.05
Q shift	0.03	0.05	0.17	0.20	0.10	0.05
MUX	0.14	0.17	0.08	0.63	0.41	0.35
ALU	0.40	0.50	0.75	0.61	0.43	0.31
Carry-propagate	0.02	0.05	0.22	0.50	0.11	0.05
Output data select	0.20	0.25	0.38	0.61	0.49	0.26
Destination control	0.10	0.14	0.10	0.57	0.20	0.10
ALU control	0.65	0.68	0.85	0.69	0.57	0.46
Source control	0.50	0.51	0.41	0.69	0.53	0.43

Table 1: Error Probability and Severity by Logic Unit

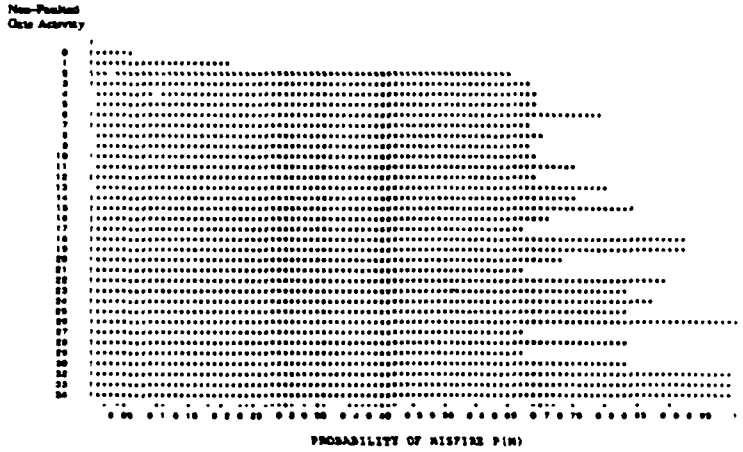
prediction, as most will agree that the control units are extremely important to the correct operation of a processor.

Under further examination, one may notice the similarity in shape between the distribution for the ALU faults and that of the overall distribution (refer back to Figure 4c). This is mainly due to the high percentage of faults injected into or near the ALU. The ALU and MUX (which feeds directly into the ALU) account for 52 percent of the total number of faults injected. Therefore, over half the simulation runs correspond to these two logic units. A point of interest is the fact that the overall distribution has smaller misfire probabilities than that for the ALU faults distribution. This difference indicates that some of the other logic units have a smaller probability of misfiring than the ALU. One explanation may be that the ALU is more highly utilized than some of the other logic units. For example, the Q shift logic unit is used mostly for the multiplication and division instructions, which are not highly used in the self-test program.

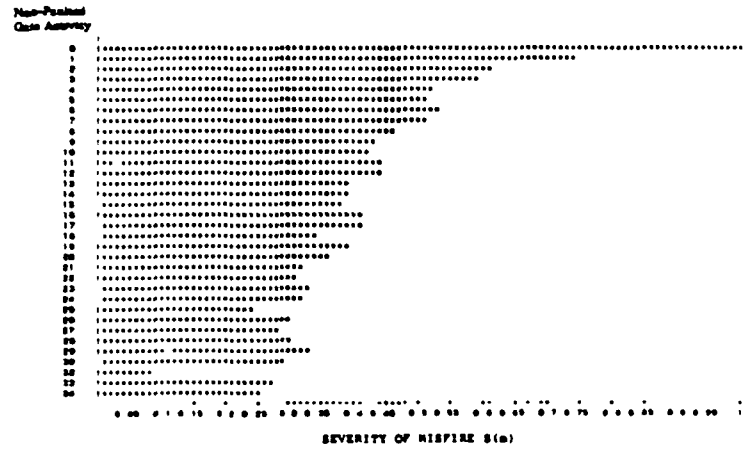
As for the effect of fault placement upon the severity of error propagation, again the control units produce the highest values, as can be seen in Table 1. (Example plots are shown in Figures 4c and 4d, again for the ALU faults and ALU control faults respectively). An explanation for this phenomenon is that faults in the control units are much more likely to affect more than one signal path. In the example of a fault in the ALU control unit, there are four data paths affected



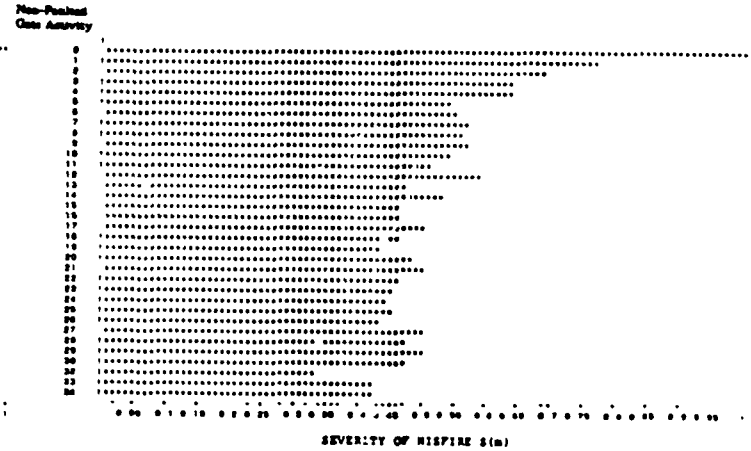
(a) For faults in the ALU



(b) For faults in the ALU control



(c) For faults in the ALU



(d) For faults in the ALU control

Figure 4: Misfire probability (MISPROB) and severity (RELSEVER) for ALU and ALU control

as opposed to only one for a fault within the ALU itself.

In summary, it was found that the faults in the ALU tend to have the characteristics of the "average" fault, i.e. the average for all 150 faults. Thus comparisons are made to this logic unit. The RAM shift and Q shift units were found to be only 0.1 times as likely to have a misfire, and similarly the severity is approximately the same. The carry-propagate unit was found to have about 0.5 times the probability/severity of the ALU unit. The MUX has comparable numbers for the severity, though the probability is much less. Finally, the control units were found to have approximately 1.5 times the probability/severity of the ALU.

7. ANALYSIS: BY INSTRUCTION/MICROINSTRUCTION

7.1. Instruction-Based Analysis

Error propagation is also highly dependent upon the assembly level instruction under execution. Thus, it is not only the amount of non-faulted gate activity which influences the error propagation, but also an interaction between gate activity and the control exerted by the instruction.

Firstly, for the same instruction, there was variation in error activity between the four samples. However, the variation between the samples was not statistically significant for identical instructions. Some variation, due to the different data operated on, even for the same type of instruction, should be expected. As the control is not changed (for the same instruction) since control was found to have the most pronounced effect upon error propagation, this observation shows the consistency of the error propagation measures used.

Comparisons are now made to see if "similar" instructions produce similar error propagation: for example, to determine whether or not two types of load instructions cause similar error behavior. For these comparisons four instructions were selected: LOAD, LDM, STO, and STM. Referring to Figures 5a through 5d (misfire probabilities for specific instructions), one can see that

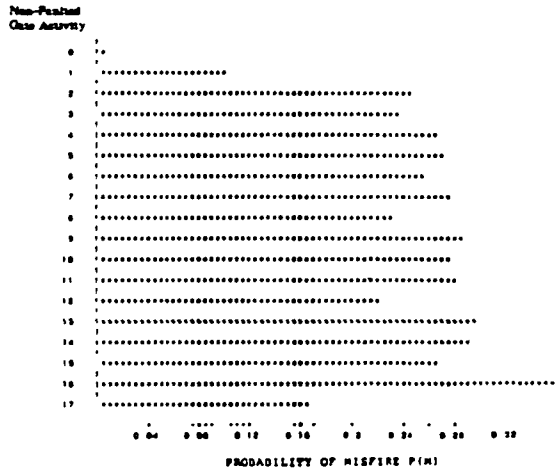
the instructions that load registers (LOAD and LDM) cause similar error behavior, as do the instructions that store register contents to memory (STO and STM). Figure 5a shows the misfire probability versus the non-faulted gate activity for the LDM instruction. Figures 5b, 5c, and 5d show similar distributions for the LOAD, STM, and STO instructions.

Now that the similarities in error propagation have been noted for similar instructions, differences between different instructions may be compared. The misfire probability distributions for the load instructions are distinct from the overall distribution (see Figure 3c which is skewed toward higher gate values).

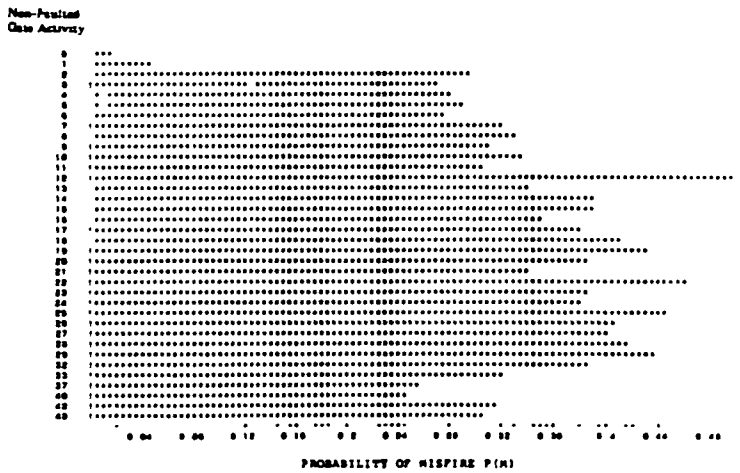
As for the severity distributions, shown in Figures 5e through 5h, again there are favorable comparisons between similar instructions. Figure 5e shows the misfire severity versus the non-faulted gate activity for the LDM instruction. Figures 5f, 5g, and 5h show similar distributions for the LOAD, STM, and STO instructions respectively. The major differences to be found are the rates at which the severity decays and the "average" level of severity. The LDM and LOAD instructions tend to maintain a higher level of severity than do the STM and STO instructions. To explain such differences, one must examine the error activity's dependency upon system activity measured at a lower level, i.e. the microinstruction level.

7.2. Microinstruction-Based Analysis

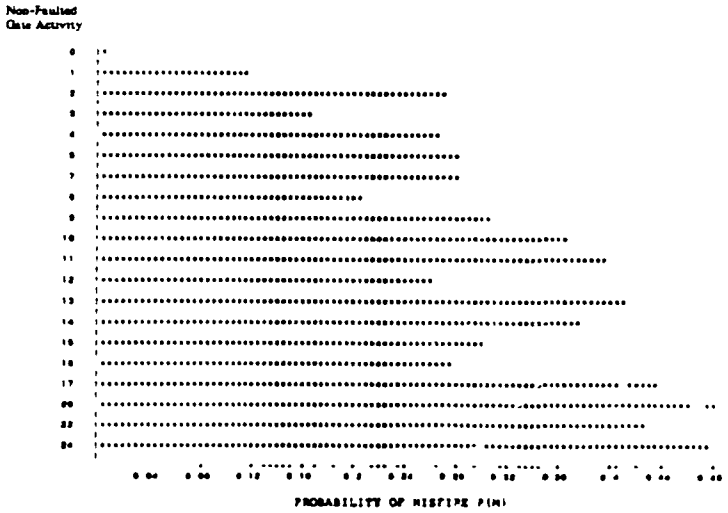
The final level of analysis is to determine the effect the type of microinstruction has upon error propagation. To this end, all of the microinstructions were classified according to the type of activity contained therein, and the error probability and error severity distribution plots were again generated. The distributions show that the probability of error activity, and the severity thereof, increase with increased microinstruction activity. Note that error activity quantified at the microinstruction level can be used to explain the error propagation at the assembly instruction level because the microinstruction is the building block of the assembly instruction.



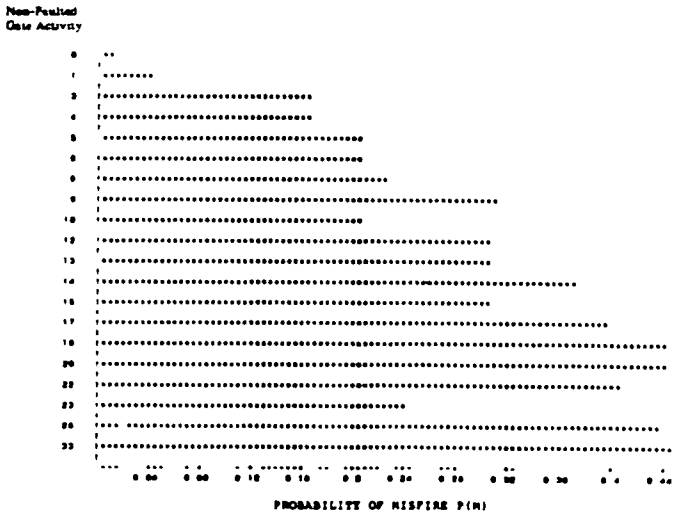
(a) For LDM instruction



(b) For LOAD instruction

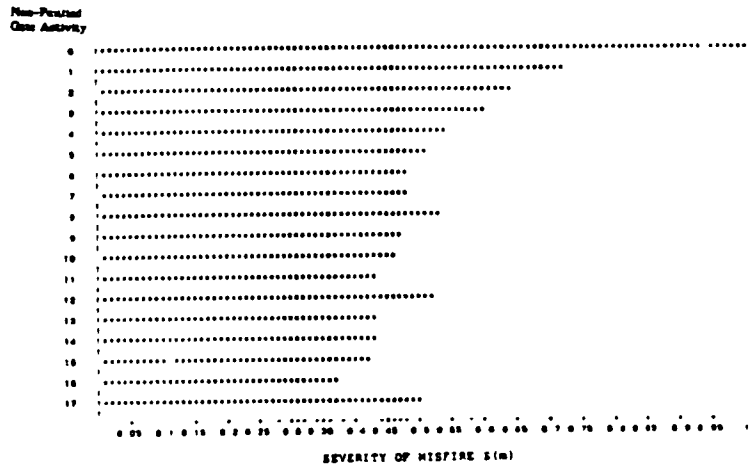


(c) For STM instruction

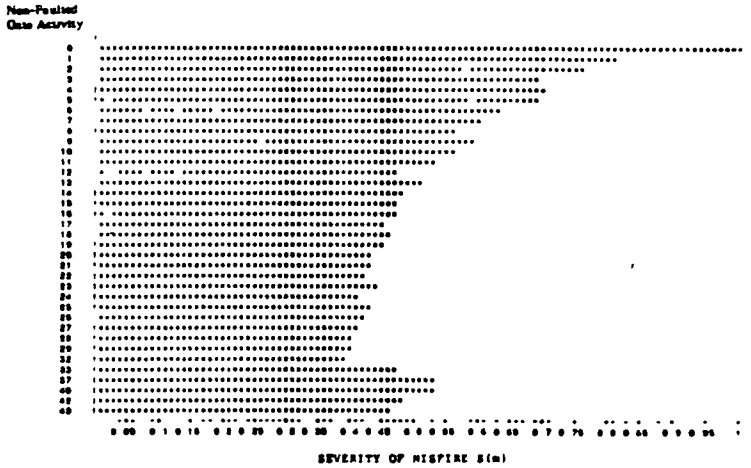


(d) For STO instruction

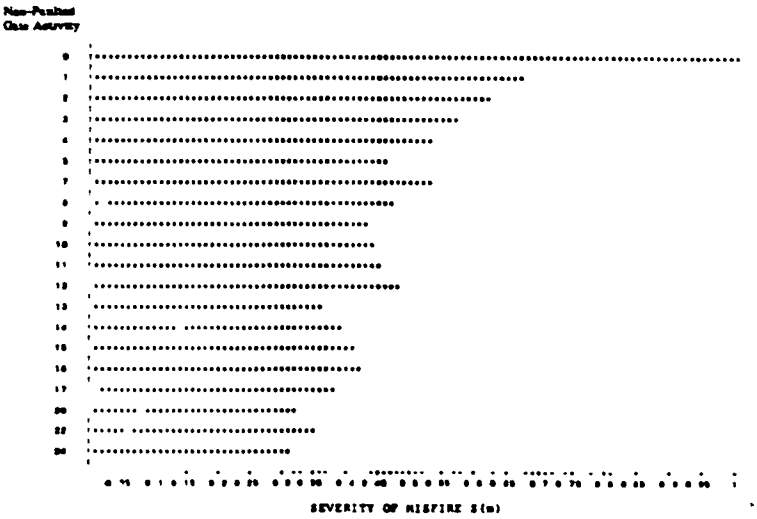
Figure 5a-d: Misfire probability (MISPROB) for LDM, LOAD, STM and STO instructions



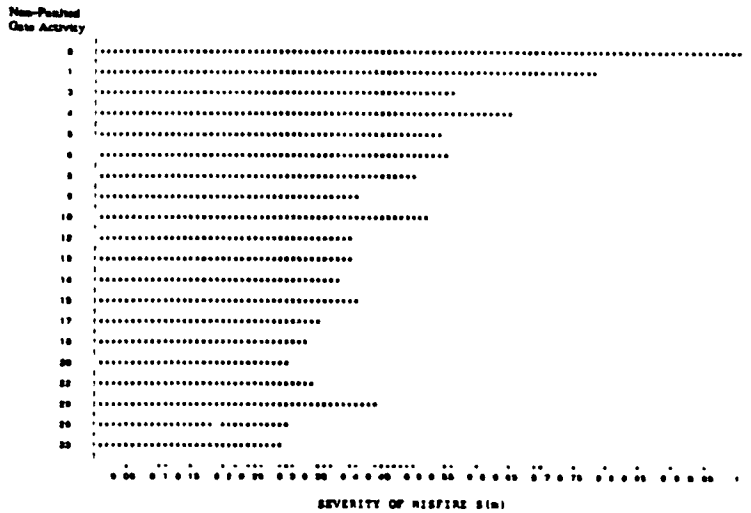
(e) For LDM instruction



(f) For LOAD instruction



(g) For STM instruction



(h) For STO instruction

Figure 5e-h: Misfire severity (RELSEVER) for LDM, LOAD, STM and STO instructions

Prior to the analysis of the error activity data according to the type of microinstruction executed, one must classify each of the microinstructions executed. The classifications defined for this study include:

Register transfer,
Memory access,
Logic computation (eg. AND, OR, etc),
Arithmetic computation, and
Conditional/unconditional branch.

Due to parallelism within the microinstructions, these classifications are not disjoint. For example, a single microinstruction may involve a register transfer and a conditional branch.

Figures 6a through 6d show that concurrent microinstruction activity implies more error activity (given a fault). For example, in Figure 6a one sees the probability of error activity for a register transfer microinstruction. When Figure 6a is compared with Figure 6b, which includes a concurrent conditional branch, it is clear that the probability of a misfire is greater with the additional activity. Similarly, the severity of error propagation is greater in Figure 6d, which contains the plot for the concurrent register transfer and conditional branch, than in Figure 6c, which contains the plot for the register transfer alone. This comparison also reinforces the above observation.

Notice also that in both Figures 6a and 6c there are deep rifts in the distributions. With the additional activity for the distributions of Figures 6b and 6d, these rifts are filled in. Since the relationship between non-faulted gate activity and error activity has already been shown for several examples, the conclusion is drawn that register transfer type microinstructions alone do not often have, for example, 6 non-faulted gate firings in a time step, whereas the combination of register transfer and conditional branch microinstructions do. Thus, it is clear that more concurrent microinstructions (more system activity) will cause more error activity. The logical explanation for this phenomenon is that concurrent microinstructions exercise the circuit more fully than does a single microinstruction.

The error propagation for various microinstruction activities is summarized in Table 2. Note that the microinstruction type 00000 is the "catch-all" i.e., it contains microinstructions that do not fit in the other classifications. The notation for the definition of the type of microinstruction is as follows:

bit_4 = 1, indicates a register transfer,
bit_3 = 1, indicates a memory access,
bit_2 = 1, indicates a logical ALU computation,
bit_1 = 1, indicates an arithmetic ALU computation, and
bit_0 = 1, indicates a conditional branch.

The relationship between concurrent microinstruction activity and error activity is quantified in Table 2. Compare the following pairs of microinstruction types: 01000 and 01001, 10000 and 10001, and also 11000 and 11001. The first type of each pair corresponds to a memory access, register transfer, and a concurrent register transfer/memory access respectively. Note that the addition of a conditional branch generally increases both the error probability and severity. The increase in severity is most often between 10 and 20 percent.

As for the difference between the store and load instructions noticed in the previous section, an explanation can be found in Table 2. The store instructions contain more memory access type microinstructions than do the load instructions. Table 2 shows that the memory access type

MICROTYPE	PROBABILITY P(M)			SEVERITY S(m)		
	50%	75%	95%	50%	75%	95%
00000	0.15	0.17	0.21	0.52	0.48	0.44
00001	0.07	0.10	0.12	0.59	0.12	0.05
01000	0.15	0.29	0.34	0.71	0.32	0.14
01001	0.23	0.32	0.45	0.65	0.45	0.33
10000	0.20	0.30	0.34	0.70	0.39	0.29
10001	0.19	0.31	0.42	0.69	0.40	0.29
11000	0.17	0.29	0.29	0.57	0.44	0.38
11001	0.23	0.27	0.40	0.61	0.46	0.34

Table 2: Error Propagation and Severity by Microinstruction Type

microinstructions have lower error severity (compare 01000 vs. 10000, and 11000 vs. 10000). This exemplifies that differences in error propagation are due to instruction type.

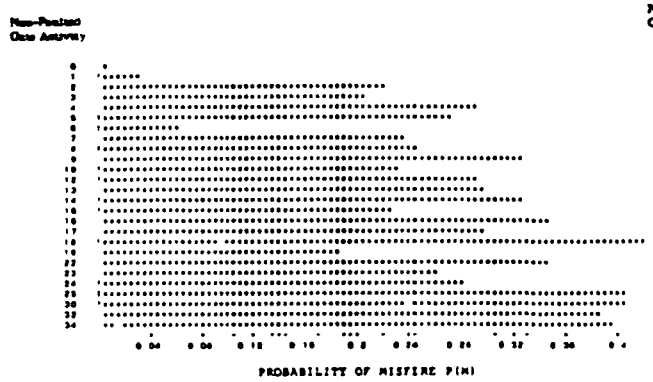
8. ERROR PROPAGATION AT THE OUTPUT PINS

Previous sections of this paper have quantified and measured error propagation throughout the AMD 2901 bit slice processor. The error activity, measured by the probability of a misfire and the severity thereof, has been quantified and also measured as a function of the number of concurrent microinstructions executed. The final determination is to characterize the error activity at the output pins and to compare the results with those obtained for the internal gates.

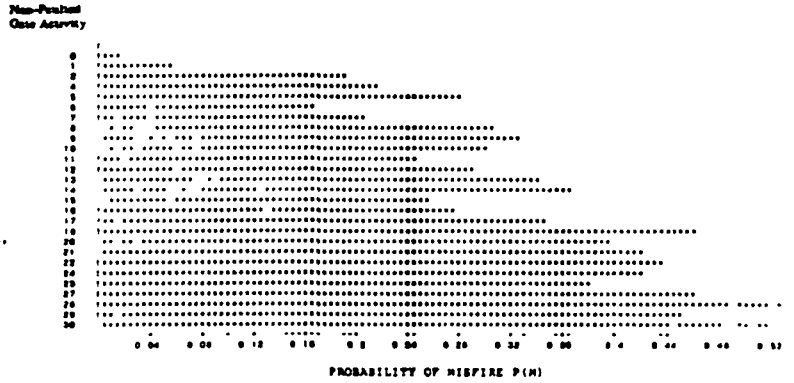
Interestingly enough, the results for the output pins are similar in the most part to those obtained for the internal gates. Hence, only a few representative plots are shown. A full range of plots is given in Appendix B. These distributions characterize the pin-level behavior corresponding to specific gate level faults.

Figure 7a shows the relationship between the probability of a misfire at one of the output pins with respect to the non-faulted output gate activity. Note that this distribution follows the trend noticed for the internal gates. The probability of a misfire is directly correlated to the non-faulted gate activity. Figure 7b, which shows the severity of the misfire at the output pins, also follows the trend for the previous data. It too shows the severity of the misfire inversely correlated to the non-faulted gate activity.

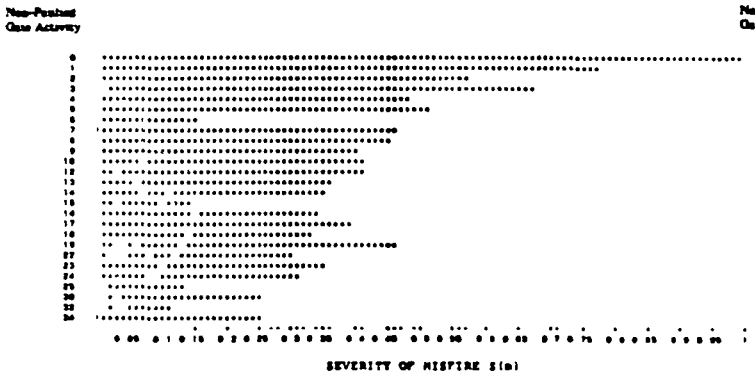
The probability of a misfire is much greater when the fault is located in the ALU control, than when it is located in the ALU itself. In fact, it was found that it was nearly twice as likely for an output misfire to occur under an ALU control fault. Similarly, the severity of the misfires for the control faults is greater. Both of these results are similar to the within-chip results. The same is true for instruction/microinstruction activity also.



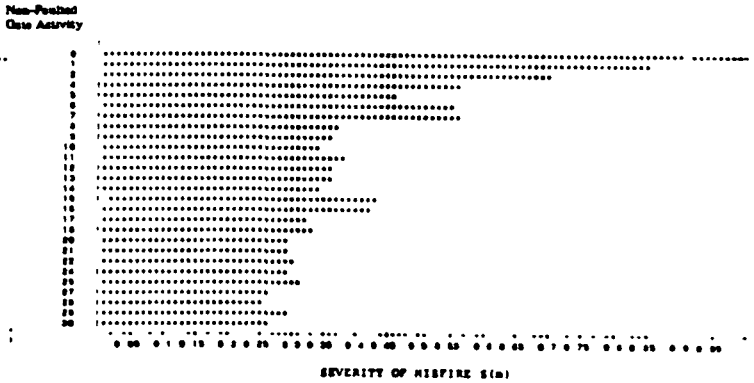
(a) For register transfer microinstruction



(b) For register transfer/conditional branch microinstruction

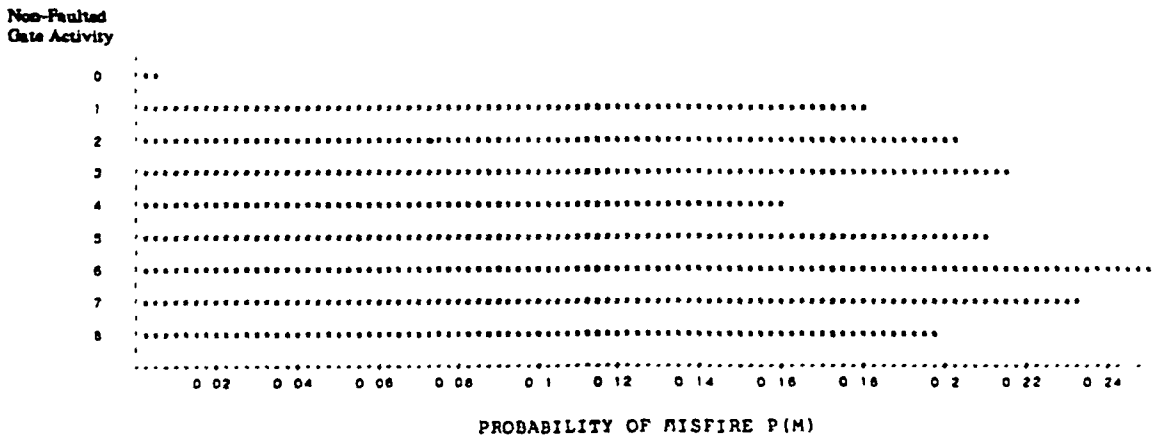


(c) For register transfer microinstruction

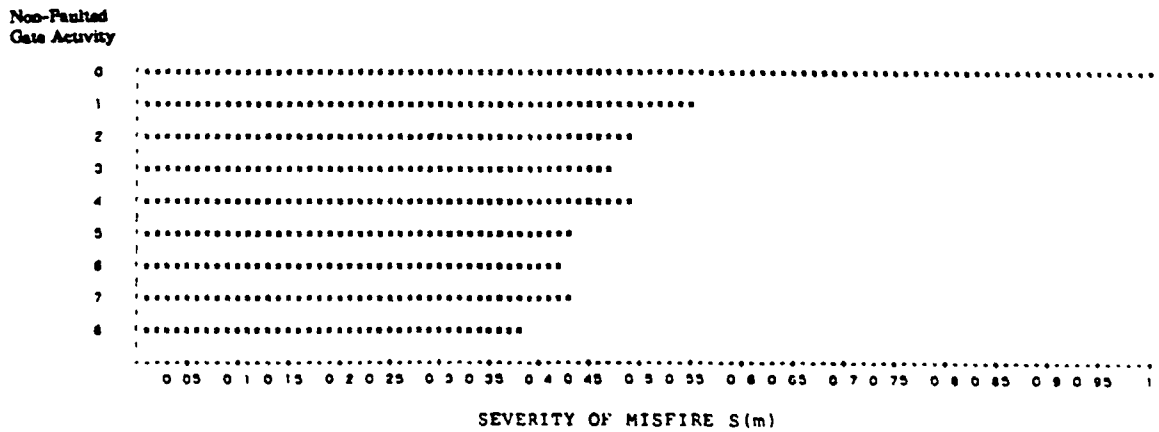


(d) For register transfer/conditional branch microinstruction

Figure 6: Misfire probability and severity by microinstruction



(a) For non-faulted case



(b) For all 150 faults

Figure 7: Output Gate misfire probability (MISPROB) and Severity

Finally, in order to determine the appropriate fault-model for pin-level faults, we plot the mean number of misfires for a given non-faulted pin activity (see Figure 8). For example with no non-faulted output activity (G = 0), a single gate-level fault in the CPU can result in (on the average) two output pin misfires in a time slice. It can be seen that most often, between two and five output pins can misfire in a time slice.

In summary, the above result strongly quantifies the inadequacy of a single fault model at the pin level. Note that a single fault model is assumed at the gate level. Further, the distributions given in Appendix B, by functional unit, instruction and microinstruction, aid in developing

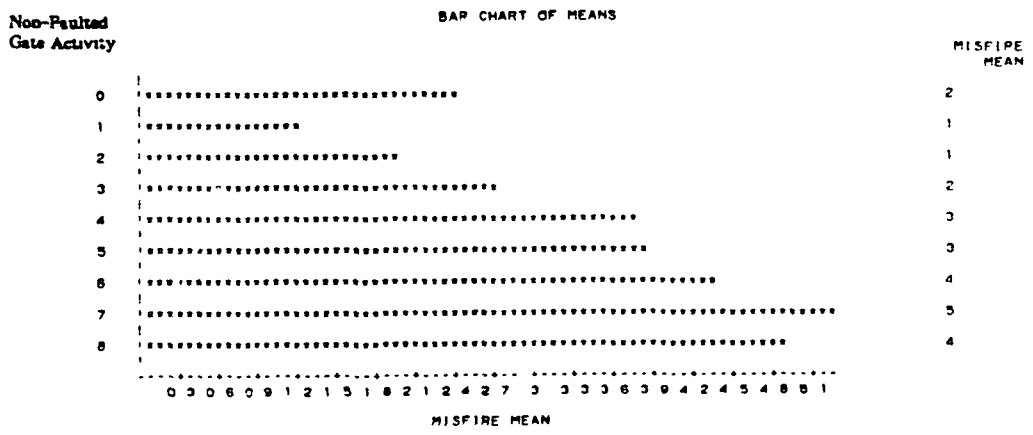


Figure 8: Mean Output Misfires per Time Step

sound pin-level multiple fault injection strategies. Using these distributions, suitable pin-level injections can now be performed to emulate specific fault behavior at the gate-level. Of course, more work is necessary to fully transform the information from these distributions into valid test strategies.

9. CONCLUSIONS

This paper has presented a methodology for the measurement and analysis of error propagation. Simulations were done for the BDX-930 digital avionic mini processor, and data was collected for the gate misfirings by finding the differences between the non-faulted master simulation run and the simulation run in which there was injected a single stuck-at fault. Measurements were taken for this data, including: histograms for the error activity over the 50 clock cycles sampled, histograms for the error activity within an "average" clock cycle, distributions for the probability of misfires, and distributions for the severity of misfires.

These measures were then examined for the effects of fault placement, type of instruction executed, and the type of microinstruction executed. Faults placed in the control units have the highest probability of causing a misfire, and if a misfire occurs, they have the highest severity. In contrast, faults placed in the RAM shift or Q shift produce very little error activity.

The error propagation is also influenced by the instruction currently under execution. Similarities in the measures were shown for similar instructions, i.e., for STO/STM and LOAD/LDM instruction comparisons. Then, differences in error propagation were noted between the power-on, store register, and load register instructions.

Finally, the influence of the type of microinstruction executed was examined. Here it was shown that concurrent microinstruction activity caused increased error activity. Furthermore, system error activity at the assembly instruction level was related to the differences in microinstruction types (concurrency in its constituents).

The analysis of the output pins showed a high degree of similarity with the results obtained for the internal gates. Importantly, it was found that a multiple fault model is necessary at the pin level to effectively describe a single stuck-at fault model at the gate level. The necessary fault distributions may be generated by gate-level fault simulations, for which a methodology is presented here.

ACKNOWLEDGMENTS

This work was supported by NASA grant NAG-1-508 and NAG-1-602. The authors wish to thank numerous people at NASA-Langley for their help on this project. Particular thanks are due to Brian Lupton, Dan Palumbo, Bernice Becher, Bob Thomas and Rudy Williams with whom we had valuable discussions. Dale Holden was particularly helpful with setting up a link to the NASA computer where the simulations were performed.

REFERENCES

[AMD81] Advanced Micro Devices. *Bipolar Microprocessor Logic and Interface Book*, Advanced Micro Devices. Sunnyvale. CA. 1981.

[Courtois79] B. Courtois. *Some Results about the Efficiency of Simple Mechanisms for the the Detection of Microcomputer Malfunctions*, Digest. FTCS-9. The Ninth International Symposium on Fault Tolerant Computing. pp. 71-74. June 20-22. 1979.

[Forman79] P. Forman and K. Moses. *SIFT: Multiprocessor Architecture for Software Implemented Fault Tolerance Flight Control and Avionics Computers*, Third Digital Avionics Systems Conference. pp. 325-329. November 6-8. 1979.

[Lala83] J.H. Lala. "Fault Detection. Isolation. and Reconfiguration in FTMP: Methods and Experiments." Fifth Dig. Avionics Syst. Conf., 1983.

[McGough81] J.G. McGough. and F.L. Swern. *Measurement of Fault Latency in a Digital Avionic Mini Processor*, NASA Contractor 3651. NASA Langley Research Center. October. 1981.

[McGough83a] J.G. McGough. *Feasibility Study for a Generalized Gate Logic Software Simulator*, NASA Contractor 172159. NASA Langley Research Center. July. 1983.

[McGough83b] J.G. McGough. and F.L. Swern. *Measurement of Fault Latency in a Digital Avionics Mini Processor Part II*, NASA Contractor 3651. NASA Langley Research Center. January 1983.

[McGough83c] J.G. McGough. Bendix; F.L. Swern. Bendix; and S. Bavuso. NASA/Langley. "New Results in Fault Latency Modeling." *Eascon, 1983.*

[Migneault85] "*The Diagnostic Emulation Technique in the Airlab*," Internal Report. NASA-Langley Research Center. 1985.

[SAS82] *SAS USER'S GUIDE: Basics*, SAS Institute Inc. Cary NC. 1982.

[Wensley78] J. Wensley et. al.. "SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control." Proc. IEEE. Vol. 66 No. 10. October 1978. pp. 1240-1254.

[Shin84a] K.G. Shin and Y.H. Lee. "*Error Detection Process - Model, Design, and its Impact on Computer Performance*," IEEE Trans. on Computers. Vol C-33. June 1984. pp. 529-540.

[Shin84b] K.G. Shin and Y.H. Lee. "*Measurements of Fault Latency: Methodology and Experimental Results*," Tech. Report CRL-TR-45-84. Computing Research Lab Univ. of Michigan. Ann Arbor. 1984.

APPENDIX A

A.1 Circuit Description

The circuit chosen for simulation is the Bendix BDX-930 digital avionic mini processor. The BDX-930 is used in a number of flight control avionics programs, notably on the AFTI F-16 FBW system and SIFT [McGough81]. Fault tolerance is achieved by replication of the processing units and voting among them in software, as in SIFT [Forman79]. In this study, only the CPU of the BDX-930 is described for simulation. The remaining portions, such as I/O and main memory are provided for by the VAX on which the simulator runs.

The actual BDX-930 consists of 86 microcircuits printed on one circuit board [Forman79]. The circuit description for simulation consists of 3212 devices. The processor is designed around the AMD 2901 four bit microprocessor slice [McGough81]. The block diagram of the CPU is given in Figure A.1. The AMD 2901 is the most complex chip in the BDX-930, and hence was chosen for the error propagation study. The simulator description of the AMD 2901 alone consists of 466 devices. There are four such chips in the BDX-930; only one was chosen for close examination. For comparison, the chip second in complexity, is the Fairchild 9407 data access register. The circuit description of this chip consists of a mere 90 devices. It seemed that this small number of devices may not allow sufficient possibility for error propagation.

A.2 Error Detection

Error propagation is detected by the comparison of two simulation runs. The first is the master or non-faulted simulation i.e., there were no faults injected. The second is with a single stuck fault injected into a single gate. By finding the differences between the outputs of these two simulation runs, one obtains a list of only the misfiring gates. This process is best understood by example.

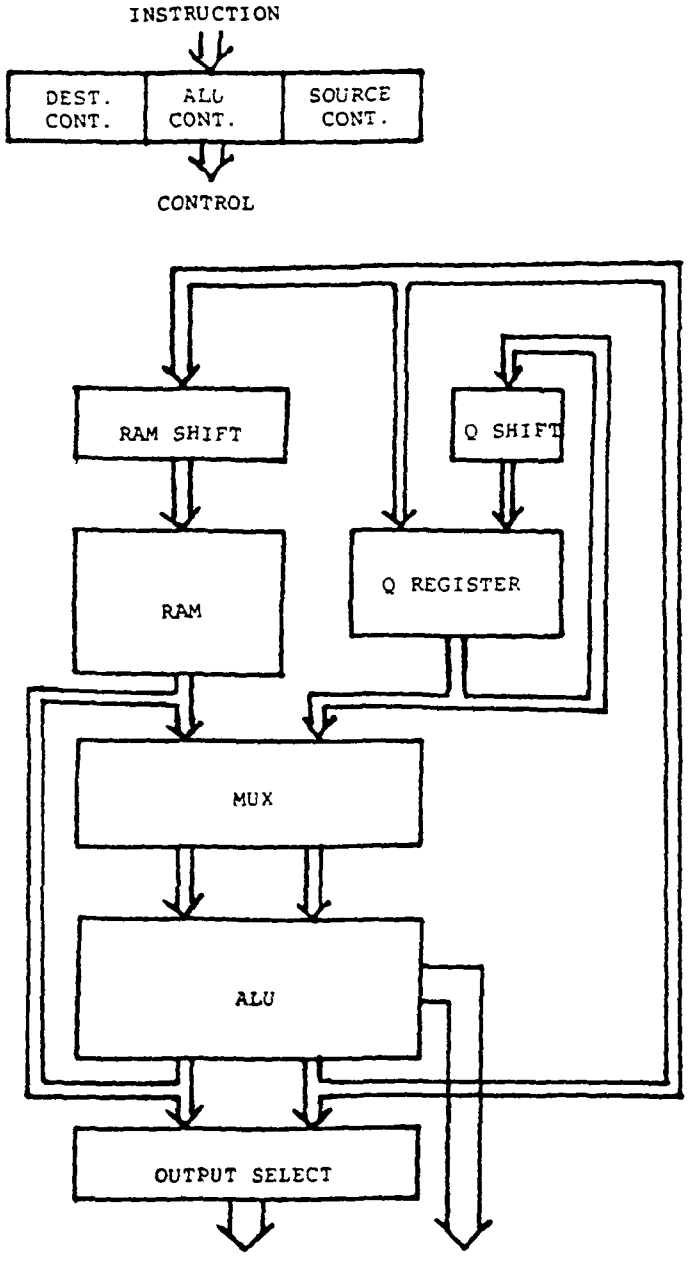


Figure A.1: Block Diagram of the AMD 2901

The first step to error propagation detection is to perform a non-faulted *master* simulation run. Table A.1 contains the gate firings for time slice 2061. The next step is to perform a "faulty" run, i.e., a simulation run obtained by injecting a stuck-at fault into a single gate. The results of a simulation for which the gate GINHCPUIC32, a gate in the ALU control logic, was stuck-at the logic value 1, are shown in Table A.2.

Having obtained the list of gate firings in the non-faulted case and the list of gate firings in the faulted case, we use the system utility VMSDIFFERENCES to obtain those gate firings that are actually misfirings. An example output of this utility may be seen in Table A.3.

The differences file is a complete list of all the misfiring gates during the time period simulated. Those gates listed in the left column are the gate firings missing in the faulty simulation run. Those gates listed in the right column are the extra gate firings in the faulty run. Those gates that appear in both columns are those that have the wrong logic value even though the gate fired in both the faulty and fault-free simulations.

Time Step	Gate Name	Gate Value
2061	GA0LCPUIC32	1
2061	GA1LCPUIC32	1
2061	GB0LCPUIC32	1
2061	GB1LCPUIC32	1
2061	TSY1CPUIC32	1
2061	TSY3CPUIC32	1

Table A-1: Gate Firings in Master Run

2061	GA0LCPUIC32	0
2061	GA1LCPUIC32	1
2061	GA2LCPUIC32	0
2061	GA3LCPUIC32	1
2061	GATE1CPUIC32	0
2061	GB0LCPUIC32	0
2061	GB1LCPUIC32	1
2061	GB2LCPUIC32	0
2061	GB3LCPUIC32	1

Table A.2: Gate Firings in Faulted Run

MISSING FIRINGS			EXTRA FIRINGS		
Time Step	Gate Name	Gate Value	Time Step	Gate Name	Gate Value
2061	GA0LCPUIC32	1	2060	GWECPUIC32	0
				2061	GA0LCPUIC32
2061	GB0LCPUIC32	1	2061	GA2LCPUIC32	0
				2061	GA3LCPUIC32
				2061	GATE1CPUIC32
				2061	GB0LCPUIC32
2061	TSY1CPUIC32	1	2061	GB2LCPUIC32	0
2061	TSY3CPUIC32	1	2061	GB3LCPUIC32	1

Table A.3: Gate Misfirings, Produced by VMSDIFFERENCES Utility

Three possible types of error propagation are detailed using this event driven simulator. Error propagation is detected when:

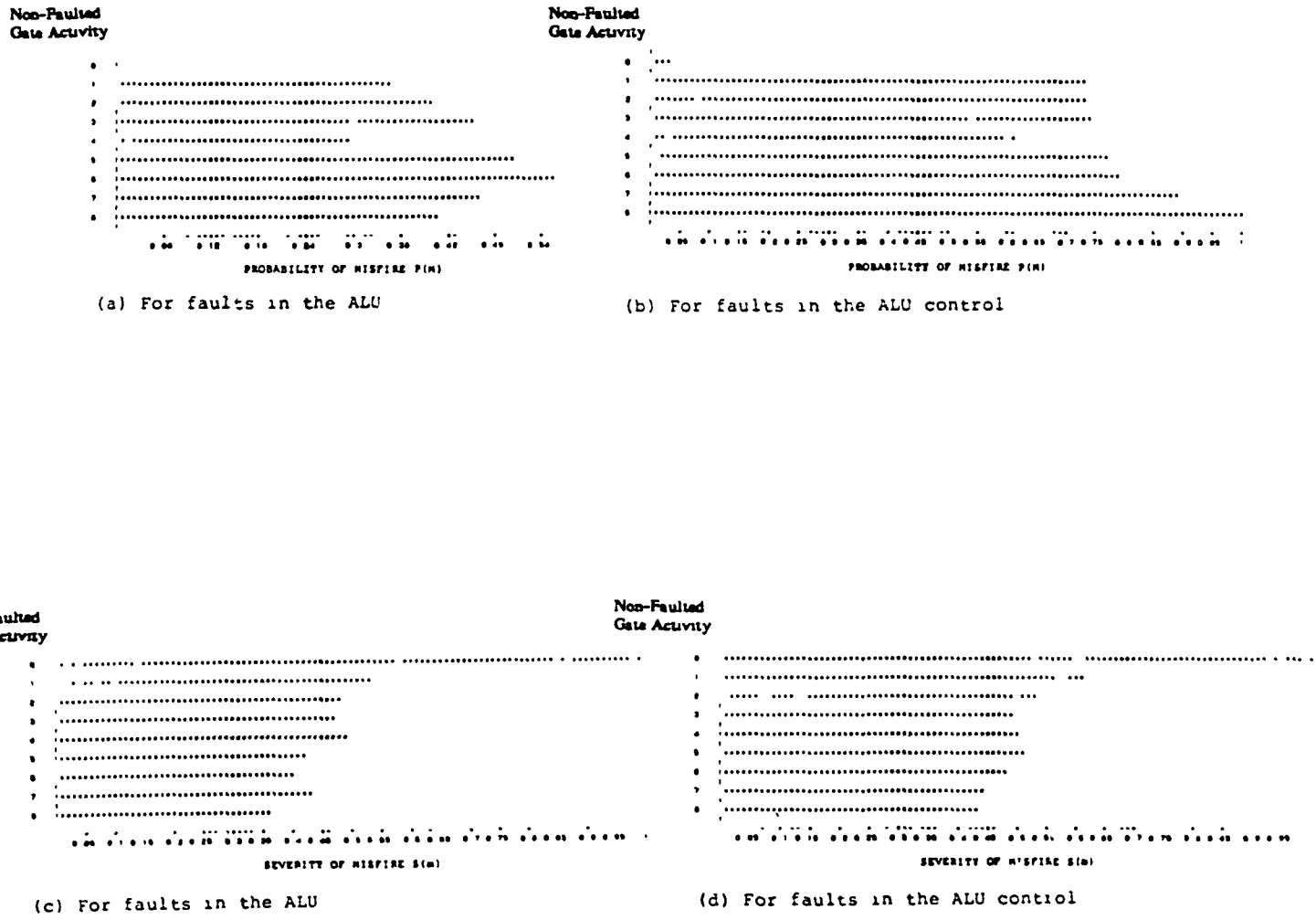
(1) there is a gate firing in the master simulation run but not in the faulty run,

(2) there is a gate firing in the faulty run that did not fire in the master run, and

(3) there is a gate firing in both runs for the same time slice but the gate takes on different logic values.

One should note that in some cases the difference file will be empty. The empty file corresponds to the fault remaining undetected, or to the fault being a latent fault, at least for the time period simulated. For the initial run of 150 individual faults, 78.7% produced error propagation detected within the chip, and 66.7% produced errors that propagated to the output pins. These results were obtained for the first 100 clock cycles, which correspond to approximately 1% of the test program. For the whole self-test program, the McGough study found a 92.0% gate-level coverage and 97.6% component-level coverage [McGough83b]. Clearly, faults tend to produce errors quickly once inserted into an active system.

APPENDIX B: Output Pin Error Distributions



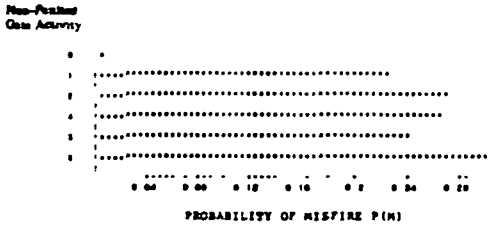
(a) For faults in the ALU

(b) For faults in the ALU control

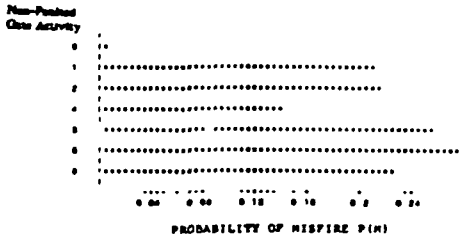
(c) For faults in the ALU

(d) For faults in the ALU control

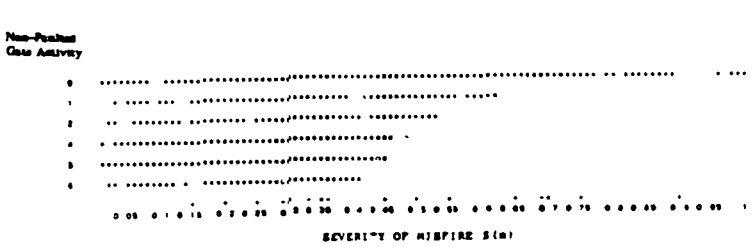
Figure B.1: Output error propagation by logic unit



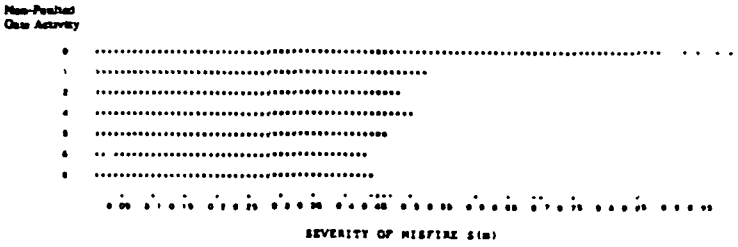
(a) For STM instruction



(b) For STO instruction

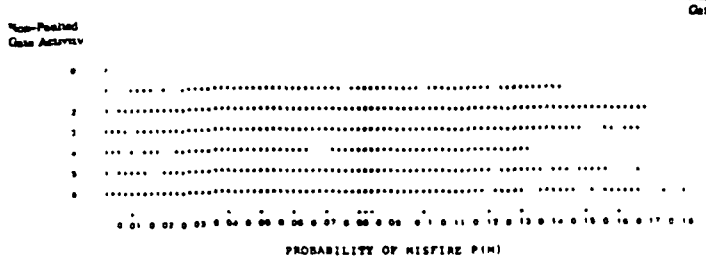


(c) For STM instruction

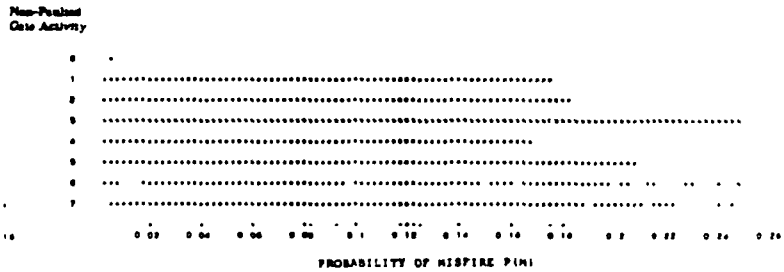


(d) For STO instruction

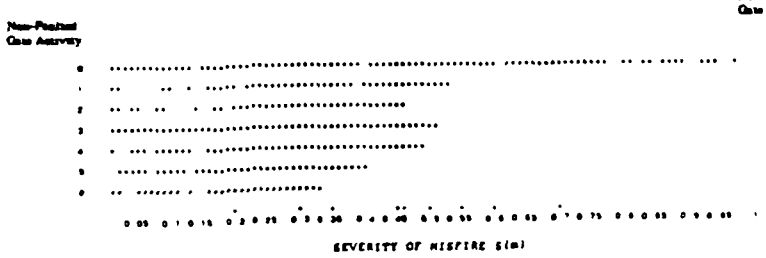
Figure B.2: Output error propagation by instruction



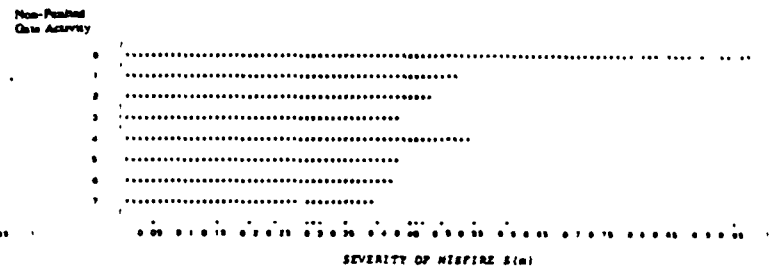
(a) For register transfer microinstruction



(b) For register transfer/conditional branch microinstruction



(c) For register transfer microinstruction



(d) For register transfer/conditional branch microinstruction

Figure B.3: Output error propagation by type of microinsruction