A Digital Logic Fault Grader

by

JOSEPH W. NAAB

BSEE, Kansas State University, 1986

A MASTER'S THESIS

submitted in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhatten, KANSAS

1988

Approved by:

[illegible]
Major Professor

A Digital Logic Fault Grader

Table of Contents

List of Figures

1. Introduction

The subject of this thesis is a C language software program that fault grades a digital circuit's input test vectors. Fault grading is the evaluation of the ability of a set of input test vectors to detect faults in a digital circuit. Inserting faults from a single stuck-at-x fault model into the circuit model, the fault grader creates a set of faulted circuit models that represent some of the physical defects and environmental effects that could occur. By simulating the unfaulted and faulted circuit models using a event driven parallel fault simulator the fault grader can predict the ability of the input vectors to detect malfunctioning circuits. The input vectors detect a faulted circuit if the outputs of a faulted circuit ever differs from the outputs of the unfaulted circuit. The fault grading techniques used by this program are discussed in Section 2 of this thesis.

This fault grading program is a student version board level fault grader. It is intended to be used in conjunction with a commercially available digital logic design package called Micro Logic 2[1]. The Micro Logic 2 package is an inexpensive, user friendly, graphics oriented package which was also designed for use at the student level. Incorporating its output files into the input files used by the fault grader was intended to make fault grading as painless as possible. The fault grader uses the netlist output of the Micro Logic 2 package to define the model of the circuit. The only modification to the Micro Logic 2 package is the addition of an 'OUTPUT' component that is used by the fault grader to identify primary outputs. The two programs were not integrated into a single package because the source code of the Micro Logic 2 package was unavailable.

Section 3 of this thesis describes the fault grading program. A description of the inputs and outputs is given along with a flow chart and general overview of the program's flow. The section also explains the interface between the Micro Logic 2 package and the fault grader in more detail, including a discussion of the addition of the 'OUTPUT' component. For more detailed information about the program, a documented copy of the C language source code has also been include in appendix B.

Element libraries are used by the fault grading program to define the elements that it is required to model. To handle a new element the libraries must be modified. Any element that appears in a circuit model that is not in the element libraries will cause the program to generate an error message and abort. The element libraries already contain all of the elements defined in the Micro Logic 2 package and expanding the libraries will usually be unnecessary. The concepts needed to expand the libraries are given in Section 4 of this thesis.

2. The Fault Grader

After the construction of a digital circuit it is necessary to determine if it works correctly. One test is to apply a set of input test vectors to the circuit and observe the circuit's outputs. If the outputs match the outputs of a 'good' circuit the circuit under test is considered a working circuit. The input test vectors used must completely test the entire circuit or a 'bad' circuit might escape the test. So a fault grader is used to determine how well the input vectors test the circuit.

The intent of the test is to cause a primary output of a 'bad' circuit to differ from the output of a 'good' circuit. A complete set of input vectors must detect the failure of each component and each connection in the circuit. Finding such a set of input vectors is not easy. Using all possible combinations of inputs is impractical for many circuits because it would take too long and if the circuit is sequential using all possible combinations of the inputs would probably still leave parts of the circuit untested.

A fault grader can be used to assist in the task of developing test vectors. It is a software program that evaluates the ability of a given set of input vectors to detect a 'bad' circuit. Models of 'bad' circuits are produced and evaluated by the fault grader. By comparing the outputs of the 'bad' circuit models with the output of a 'good' circuit the fault grader can determine which 'bad' circuits will, and will not, pass the test. Outputs generated by the fault grader signify which 'bad' circuits will be detected by each input, and which 'bad' circuits remain undetected. This information can then be used to improve the input vector set by removing the input vectors that do not detect any 'bad' circuits and writting new input vectors to test the 'bad' circuits that remain undetected.

The 'bad' circuits are modeled by faulted circuit models created with the single stuck-at-x fault model[2]. This fault model assumes that any failure in the physical circuit will look like a gate with a single input or output stuck-at either a logical 1, or a logical 0. The faulted circuit model is identical to the unfaulted circuit model except for the single fault. A complete set of stuck-at-x faulted circuits has two faulted circuits for every input and output, one modeled with a stuck-at-1 fault and the other modeled with a stuck-at-0 fault.

It is not necessary to evaluate all of these faulted circuits. Many of the faulted circuits will be indistinguishable from each other or will be automatically tested by tests for other faulted circuits. The fault grader removes these unnecessary faulted circuits in a process called fault collapsing which produces an equivalent but much smaller set of faulted circuits. A more detailed explanation and a example of fault collapsing appears in Section 2.2.

To further decrease the amount of evaluation time necessary several of the faulted circuits are simulated in parallel. This is possible because each faulted circuit only differs from the unfaulted circuit at the location of the fault. The faulted circuits must be handled separately at these points but during the rest of the simulation they are identical to the unfaulted circuit. The circuits are simulated in parallel by using each bit of the computer's simulation word for a different circuit. This simulation is almost as fast as a simple simulation being slowed slightly by the handling of the individual faults.

A event driven simulator is used to evaluate the circuits. It is an efficient simulation method because it only simulates those parts of the circuit that are active. Elements are only simulated at the point in simulation time when their outputs could change do to a previous change on one of the element's inputs. This avoids simulation of the inactive elements in the circuit and simulates the circuit dependent upon the maximum time delay of the individual elements. A more complete explanation of event driven parallel fault simulation appears in Section 2.3 and 2.4.

Two libraries provide the information that is unique to each element. The gate library is used during the model input, the fault collapsing, and the simulation processes. The gate library contains the number of inputs on each element, the time delay of the element, a function that performs fault collapsing for the gate, and a function that performs the simulation of the gate.

Another library is necessary because the Micro Logic 2 package allows the use of objects it calls 'macros' in the design of circuits. A macro is a functional unit rather than an gate level device, examples would be a SR latch or a 7474 IC chip. A macro library is used to convert faults internal to a macro to faults that occur on the macro's periphery and to remove unused parts of macros from the fault grading. The macro library contains the number of gates used in its gate level expansion, a method of identifying the different parts of a macro, and a text string for each of the possible internal faults. If an element is used in a circuit that does not appear in the element library the fault grader is forced to abort. The libraries already contain all the elements defined in the Micro Logic 2 package but it may be necessary to change the time delay information or add a new gate or a new macro so the information necessary for such modifications is included in Section 3.4.

The operation of the fault grading starts with the input of the circuit model, which is read from the gate-level network listing file produced by the Micro Logic 2 package. After forming a model of the circuit the fault grader constructs a collapsed set of faulted circuits. The fault collapsing functions in the gate library are used to collapse the faults at the individual gates. The next operation is the reading of the file that contains the input test vectors in ASCII form. If macros exist in the circuit then another file that contains the macro level representation of the circuit is read.

Once a collapsed set of faulted circuits has been constructed and the input files have been read the circuits are simulated. An event driven parallel fault simulator simulates the unfaulted circuit and as many faulted circuits as will fit in the simulation word in parallel. The simulation functions in the gate library are used to simulate the individual gates.

At the completion of the simulation of each input vector the outputs of the circuits are compared to see if any of the faulted circuits have been detected. Detected outputs statements are generated for any detected faulted circuits. After the last input vector has been simulated any of the faulted circuits in the simulation word that remain undetected will not be detected so an undetected output message is generated for each. If the fault was internal to a macro a message that explains the fault in terms of the periphery of the macro is also generated. If there are still faulted circuits unsimulated the fault grader performs the simulation process again for the next set of circuits. The outputs are discussed in more detail in Section 3.2.

2.1 The Single Stuck-At-X Fault Model

A faulted circuit model is a model of a possible failure in the circuit. A failure refers to the erroneous state of the hardware caused by any physical defect or environmental effect. Just how many such failures could exist for a given circuit is very hard to determine. Two assumptions are made to restrict the number of faulted circuits necessary to model the failures.

The first assumption will be that any failure will act like an input or an output of a gate being stuck-at either a logical 1 or a logical 0 value. This assumption simplifies the determination of the number of possible faults, because each input and output can have at most two faults, a stuck-at-1 or a stuck-at-0. The number of possible faults is given by 2n, where n is the number of inputs and outputs in the circuit. The number of possible faulted circuits is determined by the number of ways these faults can be combined, a very large number for any nontrivial circuit, leading to yet another assumption.

The second assumption will be that only one stuck-at fault can occur in a circuit at any one time. This reduces the number of faulted circuits to be exactly the number of possible faults, 2n. Each of these faulted circuits will contain only one gate which has an input or an output stuck-at either a logical 1 or a logical 0 value. The above modeling technique is commonly called the single stuck-at-x fault model, a stuck-at-x fault represents either a stuck-at-1 fault or a stuck-at-0 fault.

This fault model has been used for several years with gate level circuit representations and is still considered one of the best fault models available. Input test vectors that produce good fault coverages using the single stuck-at-x fault model catch almost all the possible defects. This is not so much a theoretical result as it is a practical one. The fault model has been widely accepted by industry to be a very good one[3].

Figure 1 shows a simple circuit and all of the faulted circuits associated with it. The single stuck-at-x fault model creates the six faulted circuits shown for the simple two input AND gate circuit. Fault grading the input vector

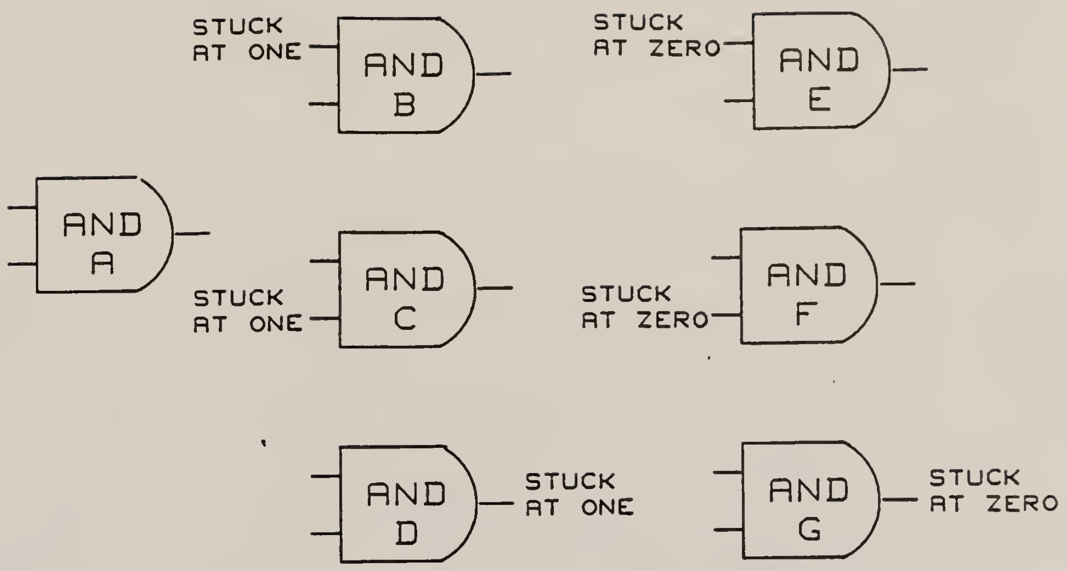

Figure 1, AND Gate and its Stuck-at-X Faulted Circuits

{1, 1} requires that all seven models be evaluated and the outputs of the six faulted circuits compared to the unfaulted circuit's output.

The unfaulted circuit outputs a 1, as would all the stuck-at-1 faulted circuits. Each of the stuck-at-0 faulted circuits outputs a 0. The result of the fault grading is 50% of the faulted circuits detected, the input vector detects exactly one half of the possible faults. Adding the vectors {0, 1} and {1, 0} to the input test vector set would result in 100% faulted circuit detection. Each of the new input vectors would detect one of the input stuck-at-1 faults and either would detect the output stuck-at-1 fault.

The fault coverage is defined as the ratio of detected faults to the number of possible faults and can be used to rate the effectiveness of the input test vectors. A good set of input vectors will give a fault coverage equal to or slightly less than one. Fault coverages of around 99% are usually considered acceptable for large complex circuits because not all the faults are detectable with a test vector set of reasonable size.

2.2 Fault Collapsing

Table 1 contains the four possible input vectors to the circuits in Figure 1, and the outputs produced by each circuit. All three stuck-at-0 faulted circuits act in the same manner for each input. This is because any stuck-at-0 fault controls the gate and forces its output to 0. The location of the stuck-at-0 fault does not matter since the three are indistinguishable, the three stuck-at-0 circuit models are equivalent. The two input stuck-at-0 faulted circuits are removed leaving five of the original seven faulted circuits.

Inputs		Outputs of Circuits							Detectable Faults	
A	B	a	b	c	d	e	f	g	SA0	SA1
0	0	0	0	0	1	0	0	0	none	d
0	1	0	1	0	1	0	0	0	none	d, b
1	0	0	0	1	1	0	0	0	none	d, c
1	1	1	1	1	1	0	0	0	e,f,g	none

Table 1, Outputs of an AND Gate and its Faulted Models.

The output stuck-at-1 fault dominates the input stuck-at-1 faults. Any test that detects one of the dominated faults, the stuck-at-1 input faults, also detects the dominating fault, the output stuck-at-1 fault. The two input vectors {1, 0} and {0, 1} each detect an input stuck-at-1 fault and they both detect the output stuck-at-1 fault. The input vector {0, 0} also detects the output stuck-at-1 fault but it detects neither of the input stuck-at-1 faults. The input vector {1, 1} detects none of the stuck-at-1 faults. Thus it is impossible to detect either of the input stuck-at-1 faults without simultaneously detecting the output stuck-at-1 fault. The output stuck-at-1 faulted model is therefore unnecessary and can be removed.

The fault grader only needs to evaluate the four remaining circuits instead of the original seven. The circuits that still must be evaluated are, the unfaulted circuit, the output stuck-at-0 faulted circuit, and the two input stuck-at-1 faulted circuits. The six original faulted circuits are represented by just the three remaining faulted circuits.

Using fault dominance and fault equivalence to reduce the number of faulted circuits leads to the following algorithm[4]:

1. Assign stuck-at-x faults to the primary inputs.
2. Assign stuck-at-x faults to the output of any gate with a fanout greater than one.
3. Translate faults forward through the circuit using equivalence and dominance.
4. Create a faulted circuit model for each remaining fault.

This algorithm produces a collapsed set of faulted circuits that is much smaller than the set of faulted circuits created by the single stuck-at-x fault model. It is still a complete set because detecting all the collapsed faults detects all possible stuck-at faults.

Calculating the fault coverage has now become difficult because no one-to-one correspondence exists between the collapsed faults and the original faults. One collapsed fault can represent any number of original faults that have been collapsed into it. The fault coverage given by the ratio of detected to possible collapsed faults should approximate the original fault coverage, at least for large circuits. This new pseudo fault coverage will be the value generated by this program.

Figure 2 is a circuit that will be used to illustrate the application of the fault collapsing algorithm. Table 2 contains the fault state of each of the inputs and outputs at each stage of the application of the fault collapsing algorithm. Each column in the table contains the set of faults that exist at each of the stages of the application of the fault collapsing algorithm.

The first step of the algorithm assigns the four primary inputs stuck-at-x faults (Stage 1, Table 2). The second step is to assign gates with fanouts greater than one, the inverters I1 and I3, stuck-at-x faults, Stage 2. The third step in the algorithm is performed repeatedly until the faults in the circuit stabilize. Feeding the faults towards the primary outputs leaves them at the inputs of the inverters and the AND gates, Stage 3. The faults at the inputs of the inverters are transferred to their outputs using fault equivalence, Stage 4. Again feeding the faults forward leaves all the faults at the AND gate's inputs, Stage 5. Using equivalence again, the stuck-at-0

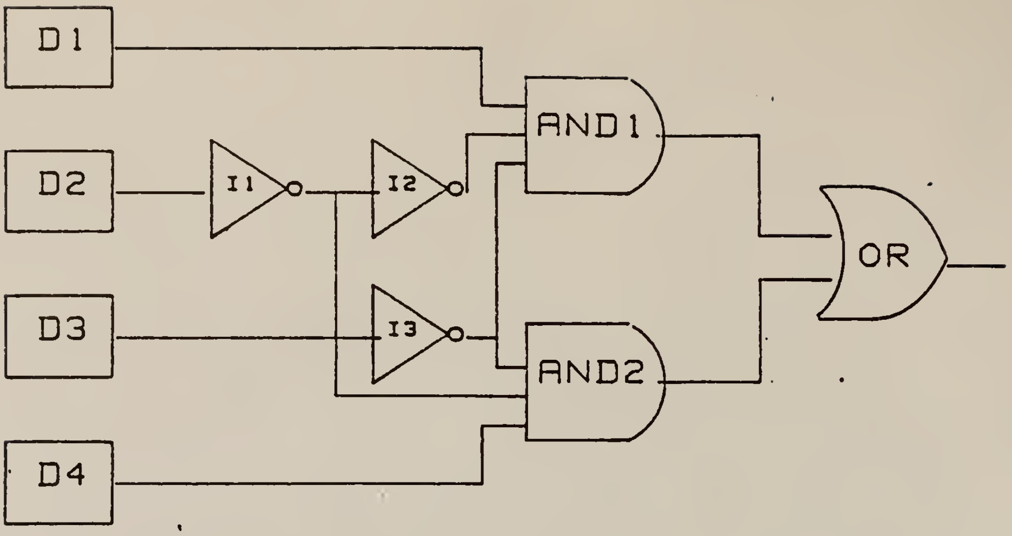

Figure 2, A Sample Circuit

Pins of the Gates	Stages 1	2	3	4	5	6	7
D1 output	SaX	SaX					
D2 output	SaX	SaX					
D3 output	SaX	SaX					
D4 output	SaX	SaX					
I1 input			SaX				
I1 output				SaX			
I2 input		SaX	SaX				
I2 output				SaX			
I3 input			SaX				
I3 output				SaX			
AND1 input			SaX		SaX	Sa1	Sa1
AND1 input2					SaX	Sa1	Sa1
AND1 input3		SaX	SaX	SaX	SaX	Sa1	Sa1
AND1 output						Sa0	
AND2 input1		SaX	SaX	SaX	SaX	Sa1	Sa1
AND2 input2		SaX	SaX	SaX	SaX	Sa1	Sa1
AND2 input3			SaX	SaX	SaX	Sa1	Sa1
AND2 output						Sa0	
OR input1							Sa0
OR input2							Sa0
total faults	8	16	16	14	12	8	8

Table 2. Fault States at Various Stages.

faults on the inputs are transferred to the outputs of the AND gates, Stage 6. Feeding the two stuck-at-0 faults forward leaves them on the inputs to the OR gate, Stage 7.

No more fault collapsing can be done and step four is performed to produce a minimum set of faulted circuits having eight members. Six of the circuits have a stuck-at-1 fault on the input of an AND gate, and two circuits have a stuck-at-0 fault on the input of the OR gate, Stage 7 again. These eight faulted circuits represent all twenty-eight of the original faulted circuits. Another useful property of the algorithm is that all the collapsed faults appear on inputs. This fact simplifies the construction of the fault grader because output faults need never be considered.

2.3 Parallel Fault Simulation

Now that a collapsed set of faulted circuits has been found it is necessary to evaluate them. The fault grader evaluates each circuit model using an event driven logic simulator. The most obvious way of performing the simulations would be to simulate the unfaulted circuit first and then each of the faulted circuits. It is faster, however, to simulate the circuits in parallel, which is possible because each faulted model differs from the unfaulted model by a single stuck-at fault. Parallel simulation of several circuits greatly reduces the amount of simulation time required.

To simulate several circuits in parallel each bit in the computer's word is assigned to a different circuit. A computer has a word length of several bits but only one of those bits is needed during simulation to represent the logical values. Another bit is required to determine if the logical value is defined or not but it can be handled in the same manner as the simulation bit. The extra bits in the simulation word are assigned faulted circuits which require special handling only at the points where a faults occur. To further increase the efficiency of the simulation, the unfaulted circuit is simulated each time, eliminating the necessity of retrieving the unfaulted outputs to compare them to the faulted outputs.

A fault simulation list contains the information for the faulted circuits that were assigned to bits in the simulation word. To simulate a faulted circuit the bit corresponding to the faulted circuit is forced to its stuck-at value at the point of the fault. The gates that have faults in the fault simulation list are marked so that as they are simulated the faults can be easily found. When a marked gate is encountered, its associated fault or faults are found in the fault simulation list so the appropriate masks can be applied.

The simulation technique just described is a common fault simulation process called, parallel fault simulation. Put concisely, a parallel fault simulator uses each bit in the computer word to simulate a different circuit in parallel. While one bit of the word is used to simulate the unfaulted circuit, each of the other bits is used to simulate a faulted circuit.

Parallel fault simulation requires a presimulation process to assign one of the faulted circuits to each bit of the simulation word. As each gate is simulated it is checked for a fault. If a fault exists in any of the circuits being simulated the bit corresponding to the fault is forced to the stuck-at fault's value. A stuck-at fault ignores the value passed to the faulted circuit's input, replacing it with the value of the stuck-at fault.

After the simulation of each individual input vector a postsimulation process looks at the values of the primary outputs. A bit that differs from the unfaulted output represents a fault that has been detected by that input vector. The detected fault is removed from the fault simulation list and an output statement identifying the fault and input vector number is generated.

A faulted circuit that never causes its corresponding simulation output bits to differ from the unfaulted circuit's output bits remains undetected. After the last input vector is simulated, any faults that remain in the fault simulation list are not detectable with the current set of input vectors. These faults are removed from the fault simulation list so the next set of faulted circuits can be simulated. As each undetected faults is removed an output statement is generated signifying that the fault remained undetected. If the fault was internal to a macro an output relating it to the periphery of the macro is also generated. When there are no faulted circuits to be simulated the parallel fault simulation process is finished.

2.4 Event Driven Simulation

The parallel fault simulation is performed using an event driven simulator which only simulates the active parts of the circuit[5]. An event driven simulator simulates events as they occur in simulation time. An event is the possible change of an element's output caused by a change to one of the element's inputs. The time at which an event will be simulated is the sum of the simulation time at which the input changed plus the element's delay time. An element's changing output causes the elements that have inputs tied to the changing output to be scheduled for simulation. An elements delay time and the simulation time used by the event driven simulator are relative figures, the only restriction is that they be integers. The initial definition of the gate library uses time measured in integer values of nanoseconds.

The event driven simulator uses a time ordered list to schedule the events. At the beginning of the simulation of each input vector the gates tied to the primary inputs are scheduled. These gates are simulated and the gates whose outputs change cause the scheduling of the gates that have inputs tied to the gate's output net. The time at which they are scheduled depends on the gate delay which is defined in the gate library. The time is calculated by adding the current time, the time when the inputs changed, to the gate delay time of the changing gate.

Gate delay times restrict the order in which the gates in the circuit are simulated. This gives a close approximation of the way the signals would actually propagate through a real circuit. An event driven simulator was used as the simulator for the fault grader because it gives a realistic simulation with respect to time and it ignores the inactive parts of the circuit.

3. The Fault Grading Program

The fault grading program is described below. First the inputs and outputs of the grader are discussed, then a general outline of the operation of the program is given along with a flow chart. The last part of this section explains the concepts necessary to expand the program to include new gates or chips. More detailed information concerning the program can be found in the documented copy of the source code in appendix B.

3.1 Input

The fault grader was designed to operate on the outputs of a commercially available digital logic design package, Micro Logic 2. The output files produced by the Micro Logic 2 package are used to define the circuit model. The first file that the fault grader reads contains the gate level network listing of the circuit. If elements more complex than simple gates were used in the design of the circuit then another network listing file is read that contains the complex elements. A complex element, which is called a macro, is a functional rather than a logical element and contains several simple gates. The last file read, which must be written by the user, supplies the input test vectors.

The three file names are, 'filename.NET', 'filenameM.NET', and 'filename.VEC'. They are accessed by the fault grader by appending the different endings to the filename which should appear on the command line that runs the program.

The network listing files contain one line for each element. Each line is numbered and contains an element name and a listing of the nets attached. Simple gates have their nets listed with the inputs first followed by the output's net. Macros have their nets listed by pin number, pin one appearing first. The listing of nets cannot begin before column eighteen. A blank line signifies the end of the model.

The input test vector file 'filename.VEC' is written by the user and is a file of ASCII 0's and 1's that defines the input test vector set. Each row of the file corresponds to a input vector and each column corresponds to an individual primary input. A value in the second column of the third row would be the value of the second primary input during the third input vector's evaluation. No spaces, blank lines or comments may appear in this file.

The CLOCK components used by the Micro Logic 2 package to clock sequential circuits are not implemented by the fault grader because the Micro Logic 2 package provides no method of outputting the clocking information. Fault grading a sequential circuit requires that one of the DATA, primary input, components be used as a clock. Two input vectors are therefore required for every different input. The first input vector sets up the values of the non-clock inputs while the clock input is a 0 and they remain the same during the second input vector which brings the clock to a 1.

The files generated by the Micro Logic package have the same base filename and are differentiated by their extensions. The network output file has an extension 'NET', the unexpanded macro file has the ending of 'M.NET', and the input vectors appear in a file with the extension 'VEC'. The outputs of the fault grader will appear in files with the same filename used by the Micro Logic package but with the extensions of 'DFA', 'UFA', and 'MFA' which are the detected fault file, the undetected fault file, and the macro fault file, respectively. The macro fault file will only appear if a macro is present in the circuit model.

3.2 Output

The fault grader generates output to both the standard output and to stored files. As faults are determined, either detected or undetected, they are sent to the standard output. Detected faults are also recorded in the file with the extension 'DFA'. The undetected faults are recorded in a file with the extension 'UFA'. If macros were used in the circuit, a macro fault file with the extension 'MFA' will also be generated. The macro fault file differs from the undetected fault file in that the undetected faults internal to macros are converted to faults on the periphery of the macros. Redundant faults that appear internal to macros are removed from the fault grading because they can not be detected by steady state testing. These faults are identified in the macro library so they can be discarded by the fault grader. The undetected faults that are not internal to a macros appear as they do in the undetected fault file.

In the output files the faults are identified by their gate number, the name of the element, the pin they appear on, and whether it is a stuck-at-1 or a stuck-at-0 fault. A gate's number and name are the same as they appear in the network file 'filename.NET'. An macro's number and name are the same as they appear in the network file 'filenameM.NET'. A detected fault's output line will also contain the number of the input vector that detected it. Undetected faults that are internal to a macro, produce two lines of output, the first maps the fault to the periphery of the macro and the second identifies the exact location of the undetected fault.

At the bottom of all three files appears the fault coverage percentage. This is calculated by dividing the number of detected faults by the total number of collapsed faults minus any redundant faults that were removed. It should be noted that this fault coverage is a relationship of the number of detected collapsed faults to undetected collapsed faults rather than a relationship of detected faults to possible faults. The two ratios are similar but differ because of the unknown number of possible faults that is represented by each collapsed fault.

3.3 Program Operation

Figure 3 is a flow chart of the fault grading program. The program begins by reading the network output file with the extension 'NET', using it to define the internal model of the circuit. If a macro is used in the circuit model the file with the ending 'M.NET' is also read. The primary inputs and the gates with fanouts greater than one are assigned stuck-at-x faults. The assigned faults are then collapsed and the first n-1 of the collapsed faults, where n is the number of bits in the simulation word, are assigned to corresponding bits of the simulation word. These faults are put on the fault simulation list and the gates they appear on are marked so the faultl can be found during simulation. The unfaulted circuit is represented with the most significant bit and the remaining bits are used to represent faulted circuits.

The input vectors are all read and the first input vector is simulated. The gates with faults are marked so that as each gate is simulated it can be checked for faults in the fault simulation list. When a marked gate is found during the simulation, the faulted circuit or circuits associated with the gate are found in the fault simulation list. Using the information in the list, the bit or bits are forced to the appropriate stuck-at value. The outputs are checked for the presence of detected faults at the completion of the simulation of each input vector. A fault is detected if its faulted circuit's primary output bits differs from the unfaulted circuit's primary output bits. Detected faults are removed from the fault simulation list and a fault detected output is generated.

The fault grader repeats the above process for each input vector. After all the input vectors have been simulated any remaining faults in the fault simulation list are undetectable with the current input vector set. These faults are removed from the fault simulation list and an undetected fault output is generated for each. The fault

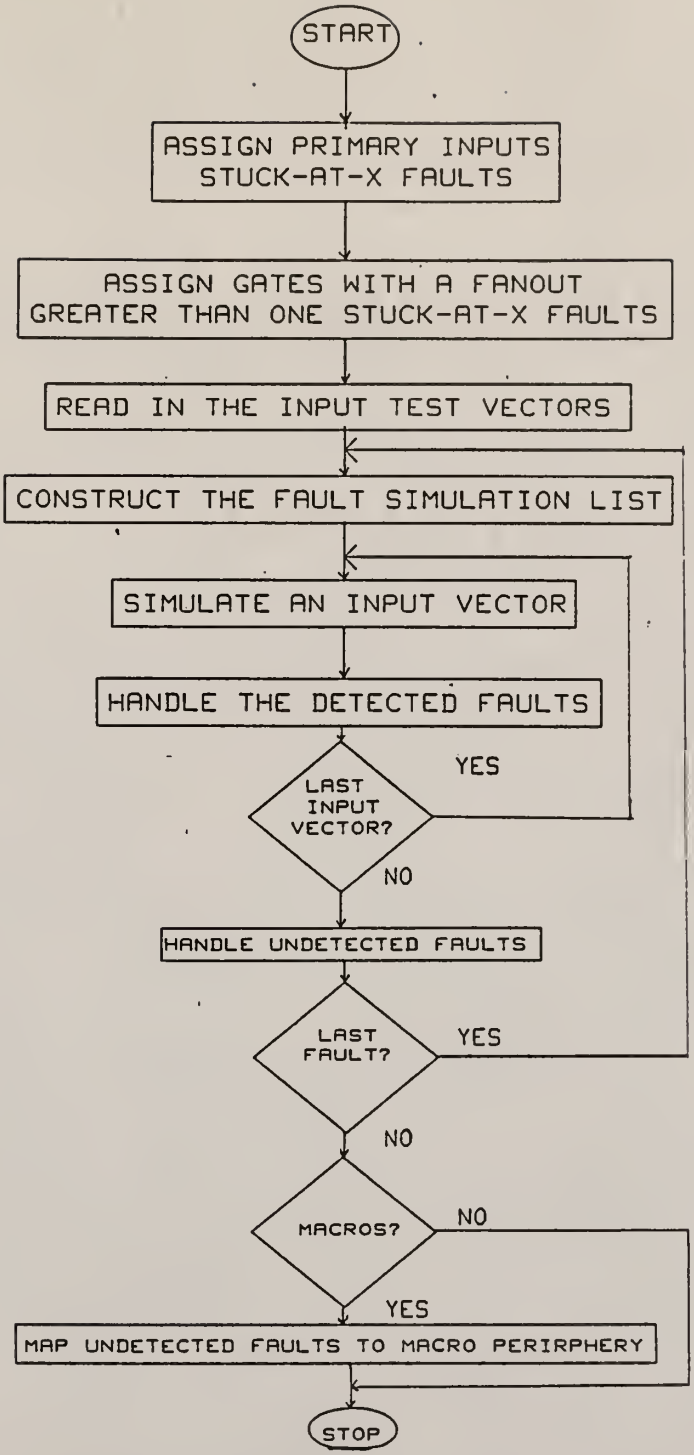
START
ASSIGN PRIMARY INPUTS STUCK-AT-X FAULTS
ASSIGN GATES WITH A FANOUT GREATER THAN ONE STUCK-AT-X FAULTS
READ IN THE INPUT TEST VECTORS
CONSTRUCT THE FAULT SIMULATION LIST
SIMULATE AN INPUT VECTOR
HANDLE THE DETECTED FAULTS
LAST INPUT VECTOR?
YES
NO
HANDLE UNDETECTED FAULTS
LAST FAULT?
YES
NO
MACROS?
NO
YES
MAP UNDETECTED FAULTS TO MACRO PERIRPHERY
STOP

Figure 3, Fault Grader's Flow Chart

grader now addresses the next n-1 faulted circuits. Faulted circuits are modeled n-1 at a time until there are no unmodeled faulted circuits remaining.

3.4 The Element Libraries

The fault grader requires considerable information about each of the elements that appear in the circuit models it grades. This information is included in element libraries which already contain all the elements that are defined in the Micro Logic 2 package, except the CLOCK component. The following section discusses the element libraries and how to add elements to them.

There are two element libraries, the gates library and the macro library. The gates are defined in a completely different manner than the macros. The information in the gates library is actually used for fault collapsing and circuit simulation, whereas the macro library is only used to interpret the network listings and the undetected faults in relation to the macro's periphery.

3.4.1 The Gate Library

The gate library is in the file 'gates.c' and consists of a structure entry and two functions for each gate. The structure is initialized with the gate's name, the number of pins, the time delay of the gate, the name of the function used for fault compression, and the name of the function used for simulation. An example initialization line is given by Figure 4. The name of the gate can not exceed eight characters because of the input specifications of the fault grader. A gate can have any number of pins up to sixteen, but only one of them may be an output. The time delay of the gate must be an integer and is used by the simulator to determine at what time the gate's output will change after an input value changes.

```
"AND2", 3, 1, and2f, and2s,
```

Figure 4. Sample 'GATES' Initialization Line.

The function names must be the names of valid C user-supplied functions. The fault collapsing function is used during the fault compression stage to transfer faults from the inputs of the gate to the its output, using equivalence and dominance rules. For a two-input AND gate this corresponds to transferring any stuck-at-0 faults from the inputs to the output. Figure 5 is the fault collapsing function for a two-input AND gate. The function first looks at the output of the AND gate for a stuck-at-0 fault. If one exists then any stuck-at-0 input faults are removed and the status of the gate's output is returned as unchanged. If no stuck-at-0 exists on the output then the function looks at the inputs to see if there are any input stuck-at-0 faults. If there is not then the gate's output status is again returned as unchanged, but if there is a input stuck-at-0 fault an output stuck-at-0 fault is produced, any input stuck-at-0 faults are removed and the output status of the gate is returned as having acquired a stuck-at-0 fault.

```
int and2f( state )

  int *state;

{
  if( !( *state & 0x10 ))
    if( *state & 0x5 )
    {
      *state |= 0x10;
      *state &= 0x3a;
      return 1;
    }
  *state &= 0x3a;
  return 0;
}
```

Figure 5, An AND Gate's Fault Collapsing Function.

This function can be used as a template for the development of a fault collapsing function for a new gate. The argument 'state' represents the faults by using each bit of the word to represent one of the possible faults. The least significant two bits represent the faults on the first input, the next two bits represent the next input and so on until the output's faults are represented. The least significant bit in each pair represents the stuck-at-0 fault. The other bit represents the stuck-at-1 fault. A fault exists if the bit representing it is set.

The 'and2f' is the name of the function and is the only item in the first five lines that should be modified. The next part of the function that differs from gate to gate is the conditional of the first 'if', line 6. The conditional checks to see if the output is already faulted. Bitwise anding the word 'state' with the hexadecimal value 10 will be nonzero, or true, only if the gate has a stuck-at-0 fault on its output. If this is true the output fault state of the AND gate will not change regardless of the faults input into the gate. The '!' is a NOT operator and changes the tested condition from true to false. The 'if' fails so the program flow skips down to line 13 which masks off any input stuck-at-0 faults. A stuck-at-0 fault on the output is equivalent to the input stuck-at-0 faults so the input stuck-at-0 faults are removed.

If the conditional is true, no stuck-at-0 on the output, line 7 is processed. The 'if' on line 7 tests for the presence of a stuck-at-0 fault on either input. If there is not an input stuck-at-0 fault the program skips to line 13 as described above. If there is an input stuck-at-0 fault, the output is given a stuck-at-0 fault, line 9. Any input stuck-at-0 fault is removed and a value of 1 is returned, lines 10 and 11. The value 1 is returned to signify that the output fault state of the gate has changed and that it has acquired a stuck-at-0 fault. Returning a value of 2 would indicate that the output had acquired a stuck-at-1 fault, and a returned value of 3 would represent both faults. A value of 3 can only be returned by elements that simply transfer faults, like an inverter. It is necessary to return the new fault value of the output so it can be propagated forward through the circuit.

The simulation functions are used to simulate the gates. The simulation function for the two-input AND gate is shown in Figure 6. This function uses the inputs to the AND gate to determine the output of the gate. Two variables are required to define a gate's output, the logical value of the output and whether or not the output is defined. The output of the AND gate is defined if either of the inputs has a defined logical 0 value or if both of the inputs have defined logical values. The logical output of the AND gate is simply the logical anding of the inputs regardless of their defined states.

```
int and2s( logic_input, define_input,
           logic_output, define_output )

long *logic_input,
     *define_input,
     *logic_output,
     *define_output;

{
  *define_output = ( define_input[0] & ~logic_input[0] )
                 | ( define_input[1] & ~logic_input[1] )
                 | ( define_input[0] & define_input[1] );

*logic_output = ( logic_input[0] & logic_input[1] );
}
```

Figure 6, An AND Gate's Simulation Function.

The arguments input to all the simulation functions are the same. The first two are pointers to integer arrays that hold the values of the inputs. The array element indexed with a 0 refers to the first pin, the array element indexed with a 1 refers to the second input, and so on for each of the inputs. The array pointed to by the pointer 'logic_input' contains the logical values of the inputs. The array pointed to by 'define_input' contains the defined states of the inputs. A 0 bit in the 'define_input' array means that the logical value for that input is not defined.

The simulation function only has two statements. The first one, which starts on line 10, determines the output defined state. It states that, if either the first or the second input has a defined logical value of 0, or if both inputs are defined, then the output is defined. The second statement, line 14, determines the value of the logical output regardless of the defined states.

The operations performed on the inputs and outputs must be bitwise. This is because each bit of the simulation word represents a different circuit. The '&' operator AND's the bits of its operands, the '|' OR's the bits of its operands, and the '~' complements each bit of its operand.

The steps for adding a gate to the library file 'gates.c' are:

- Write a fault function for the gate.
- Write a simulation function for the gate.
- Add the gate to the initialization list of the structure 'GATES'.
- Increase the value of 'GATE_NUMBER' by one for each gate added to the library.
- Recompile and relink the file 'gates.c'.

3.4.2 Macro Library

The macro library appears in the file 'macros.c' and contains the information necessary to relate faults to the macros used in the circuit model and to remove the unused parts of macros from the fault grading. A macro is a collection of gates grouped together to form a functional unit or to model a chip level device. The purpose of the macro library is to map undetected faults internal to the macro to their corresponding peripheral faults. The intent is to give the program user a idea of the faults remaining untested on the periphery of the macro. The library also supplies information that is used for the removal of unused macro faults and redundant faults internal to a macro. Faults that are undetectable, e.g. because of redundancy in the circuit, are marked so they can be ignored by the program.

The best method for finding the faults associated with a new macro is to individually fault grade its expanded form. The fault list generated by this method is a complete fault list that contains all the macro's internal faults. Once the relationships between the internal faults and their corresponding peripheral faults are known, the macro can be added to the macro library.

When a macro is used to represent a chip level device it is possible that extra unused gates will be included in the model. Some chips are dual devices, there are two identical functions provided by the chip one of which could remain unused. The macro library provides enough information so that the unused parts of a macro can be removed from the fault grading.

Three structures and an integer array are used in the macro library. The first structure 'macros' contains the characteristics of the macro in one initialization line for each macro. The line contains the macro name, the number of gates in the macro's expansion, the number of faults defined for the macro, an index into the second structure, 'macrofaults', and a index into the third structure, 'macroparts'. The macro's name is a character string identical to the string that will identify the macro in the Micro Logic 2 netlist output. Figure 7(a) gives a sample initialization line for a TTL 7474 chip.

The second structure holds the statements that map the internal faults to the periphery of the macro. It requires one initialization line per possible fault, which contains the gate and pin the fault is located on, the type of fault, and an output statement for the fault. The gates

```
"74", 12, 36, 0, 1
```

(a)

```
1, 1, 1, "Preset stuck-at-1 fault",
```

(b)

```
7, 1, 6
```

(c)

Figure 7, Macro Library Initialization Lines.

and pins are numbered sequentially as they appear in the macro expansion produced by the Micro Logic 2 package. The type of the fault is given by a '1' if it is a stuck-at-1 and a '0' if it is a stuck-at-0. The output statement will be printed immediately preceding the undetected fault statement with the intention of explaining the internal fault. The convention in the macro library is that this message be mnemonic in nature but should contain the pin number when appropriate. Figure 7(b) gives a sample initialization line.

The third structure holds three integers that are used to identify the parts of compound macros. Macros that are not compound, i.e. have only one functional unit, all use the first line of this structure. The first two integers specify a gate and a pin, respectively, that is checked to see if the macro part is in use. If the pin is tied to the 0 net, the null net, then that part of the macro is not used by the circuit. The third integer is the number of gates used to expand the macro part. When an unused macro part is found the gates used in that macro parts expansion have their faults removed, this removes the unused macro part from further consideration by the fault grader. The removed faults do not appear in the fault grader's outputs, nor are they included in the calculation of the fault coverage.

The steps for adding a macro to the library file 'macros.c' are:

- Create an initialization line in the structure MACROS.
- Create an initialization line in the structure MACROFAULT for every possible internal macro fault.
- If the macros has multiple parts, create a initialization line in the structure MACROPARTS for each part.
- Increase the value of MACRO_NUMBER one for each new macro.
- Recompile and relink the file 'macros.c'.

4. Outputs and Results

The circuit in Figure 8(a) is a five state sequential circuit which will be used to illustrate how the fault grader operates. The circuit was chosen as a simple and traceable circuit that will completely exercise the fault grader. A state diagram of the circuit's operation, Figure 8(b), shows that the circuit does little more than walk around in a circle, occasionally taking a shortcut dependent upon the two inputs.

The two listings in Figure 9 are the netlist outputs of the Micro Logic 2 package after it was used to make the schematic of the circuit. They are the files that the fault grader will use to define its circuit model. The first listing, Figure 9(a), is the unexpanded macros netlist and is the most readable. The second, Figure 9(b), is the one the fault grader uses most. In this file the macros of the D flip-flops have been expanded to the gate level. The fault grader does all its work at the gate level, referring to macros only for its I/O interfacing.

Figure 10 shows the input vectors as they are read by the fault grader. This file is not created by the Micro Logic 2 package, but it is produced by the user, using any text editor. The columns in the file correspond to the 'DATA' inputs, the first column is the first data input, 'DATA1'. The rows in the file correspond to the input vectors, the first row is the first input vector. The fault grader does not have a clock input so one of the data inputs must be

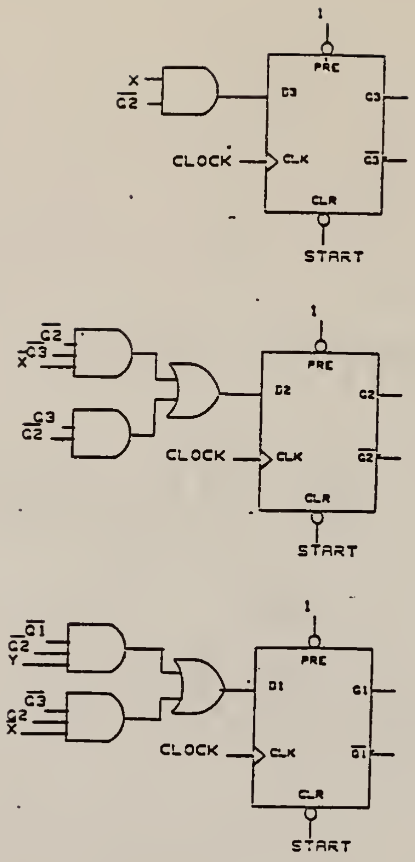

Figure 8(a), A Five-State Sequential Circuit

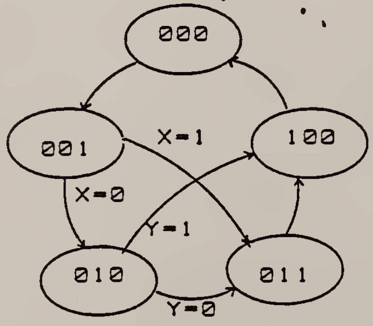

Figure 8(b), Five-State Circuit's State Diagram

```
                                     Report for STATEM
 1     DATA1       1
 2     DATA2       2
 3     AND2        1 13 3
 4     OR2         5 6 4
 5     AND2        14 16 5
 6     AND3        10 13 15 6
 7     AND3        9 14 18 7
 8     OR2         7 5 8
 9     DATA3       9
 10    INV         1 10
 11    DATA4       11
 12    DATA5       12
 13    OUTPUT      18 0
 14    OUTPUT      13 0
 15    OUTPUT      14 0
 16    OUTPUT      17 0
 17    OUTPUT      15 0
 18    OUTPUT      16 0
 19    74          11 4 2 12 13 14 0 15 16 12 2 8 11 0
 20    74          11 3 2 12 17 18 0 19 20 0 0 0 0 0
```

Figure 9(a), Five-State Circuit's Unexpanded Netlist

used. The other data inputs should be stable, unchanged as the clock's data input rises from '0' to '1' or the timing requirements of the D flip-flop will be violated in which case the simulator does its best to simulate the circuit but it was not designed to handle races.

The three files that are produced by the fault grader are shown in Figure 11. They are the detected fault file, Figure 11(a), the undetected fault file, Figure 11(b), and the macro mapped undetected fault file, Figure 11(c). Each of the files contain the fault coverage percentage calculated from the collapsed fault set. All three types

```
                                     Report for STATE
1     DATA1       1
2     DATA2       2
3     AND2        1 13 3
4     OR2         5 6 4
5     AND2        14 16 5
6     AND3        10 13 15 6
7     AND3        9 14 18 7
8     OR2         7 5 8
9     DATA3       9
10    INV         1 10
11    DATA4       11
12    DATA5       12
13    OUTPUT      18 0
14    OUTPUT      13 0
15    OUTPUT      14 0
16    OUTPUT      17 0
17    OUTPUT      15 0
18    OUTPUT      16 0
19    NAND3       12 21 22 23
20    NAND3       23 11 2 22
21    NAND3       22 2 21 24
22    NAND3       24 11 4 21
23    NAND3       12 22 14 13
24    NAND3       13 11 24 14
25    NAND3       12 25 26 27
26    NAND3       27 11 2 26
27    NAND3       26 2 25 28
28    NAND3       28 11 8 25
29    NAND3       12 26 15 16
30    NAND3       16 11 28 15
31    NAND3       12 29 30 31
32    NAND3       31 11 2 30
33    NAND3       30 2 29 32
34    NAND3       32 11 3 29
35    NAND3       12 30 18 17
36    NAND3       17 11 32 18
37    NAND3       0 33 34 35
38    NAND3       35 0 0 34
39    NAND3       34 0 33 36
40    NAND3       36 0 0 33
41    NAND3       0 34 19 20
42    NAND3       20 0 36 19
```

Figure 9(b), Five-State Circuit's Expanded Netlist

```
10000
10011
11011
10111
11111
00011
01011
00111
01111
```

Figure 10, Five-State Circuit's Input Vectors

of output messages are sent to the standard output as the fault grader runs, ordered by the their occurrence in time only.

The fault coverage value gives an idea of how well the circuit was tested, but it does not help in the creation of better test vectors. To improve the test one must look at the undetected fault files and try to create additional inputs that will detect those faults. It may also be possible to decrease the size of the test vectors by looking at the detected fault file and removing the input vectors that do not detect any faults. But care must be taken with sequential circuits, in order that initialization input vectors are not removed.

A macro appeared in the circuit so the macro mapped undetected fault file may also be useful because it contains output statements that relate the internal faults

```
23 NAND3     pin 1 stuck-at-1 detected by input   1
24 NAND3     pin 2 stuck-at-1 detected by input   1
20 NAND3     pin 3 stuck-at-1 detected by input   2
24 NAND3     pin 1 stuck-at-1 detected by input   2
 3 AND2      pin 2 stuck-at-1 detected by input   3
 5 AND2      pin 2 stuck-at-1 detected by input   3
20 NAND3     pin 1 stuck-at-1 detected by input   3
22 NAND3     pin 3 stuck-at-1 detected by input   3
 7 AND3      pin 1 stuck-at-1 detected by input   3
22 NAND3     pin 1 stuck-at-1 detected by input   5
 8 OR2       pin 1 stuck-at-0 detected by input   5
 4 OR2       pin 1 stuck-at-0 detected by input   7
19 NAND3     pin 2 stuck-at-1 detected by input   7
19 NAND3     pin 3 stuck-at-1 detected by input   7
21 NAND3     pin 2 stuck-at-1 detected by input   7
23 NAND3     pin 2 stuck-at-1 detected by input   7
21 NAND3     pin 1 stuck-at-1 detected by input   7
 8 OR2       pin 2 stuck-at-0 detected by input   7
23 NAND3     pin 3 stuck-at-1 detected by input   8
 3 AND2      pin 1 stuck-at-1 detected by input   9
 5 AND2      pin 1 stuck-at-1 detected by input   9
 6 AND3      pin 3 stuck-at-1 detected by input   9
 7 AND3      pin 2 stuck-at-1 detected by input   9
35 NAND3     pin 1 stuck-at-1 detected by input   1
30 NAND3     pin 2 stuck-at-1 detected by input   1
29 NAND3     pin 1 stuck-at-1 detected by input   1
32 NAND3     pin 3 stuck-at-1 detected by input   2
26 NAND3     pin 3 stuck-at-1 detected by input   2
30 NAND3     pin 1 stuck-at-1 detected by input   2
32 NAND3     pin 1 stuck-at-1 detected by input   3
26 NAND3     pin 1 stuck-at-1 detected by input   3
28 NAND3     pin 3 stuck-at-1 detected by input   3
25 NAND3     pin 2 stuck-at-1 detected by input   5
27 NAND3     pin 2 stuck-at-1 detected by input   5
29 NAND3     pin 2 stuck-at-1 detected by input   5
29 NAND3     pin 3 stuck-at-1 detected by input   6
25 NAND3     pin 3 stuck-at-1 detected by input   7
27 NAND3     pin 1 stuck-at-1 detected by input   7
24 NAND3     pin 3 stuck-at-1 detected by input   9
28 NAND3     pin 1 stuck-at-1 detected by input   9
30 NAND3     pin 3 stuck-at-1 detected by input   9
36 NAND3     pin 2 stuck-at-1 detected by input   1
36 NAND3     pin 1 stuck-at-1 detected by input   2

The fault coverage was 64.18%.
```

Figure 11(a), Detected Fault File

```
  4 OR2       pin 2 stuck-at-0  remains undetected.
  6 AND3      pin 1 stuck-at-1  remains undetected.
  6 AND3      pin 2 stuck-at-1  remains undetected.
  7 AND3      pin 3 stuck-at-1  remains undetected.
 19 NAND3     pin 1 stuck-at-1  remains undetected.
 20 NAND3     pin 2 stuck-at-1  remains undetected.
 21 NAND3     pin 3 stuck-at-1  remains undetected.
 22 NAND3     pin 2 stuck-at-1  remains undetected.
 25 NAND3     pin 1 stuck-at-1  remains undetected.
 26 NAND3     pin 2 stuck-at-1  remains undetected.
 27 NAND3     pin 3 stuck-at-1  remains undetected.
 28 NAND3     pin 2 stuck-at-1  remains undetected.
 31 NAND3     pin 1 stuck-at-1  remains undetected.
 31 NAND3     pin 2 stuck-at-1  remains undetected.
 31 NAND3     pin 3 stuck-at-1  remains undetected.
 32 NAND3     pin 2 stuck-at-1  remains undetected.
 33 NAND3     pin 1 stuck-at-1  remains undetected.
 33 NAND3     pin 2 stuck-at-1  remains undetected.
 33 NAND3     pin 3 stuck-at-1  remains undetected.
 34 NAND3     pin 1 stuck-at-1  remains undetected.
 34 NAND3     pin 2 stuck-at-1  remains undetected.
 35 NAND3     pin 2 stuck-at-1  remains undetected.
 35 NAND3     pin 3 stuck-at-1  remains undetected.
 36 NAND3     pin 3 stuck-at-1  remains undetected.
The fault coverage was 64.18%.
```

Figure 11(b), Undetected Fault File

to the periphery of the macro. Not all of the internal faults possible in a macro can be directly mapped to a macro pin. Some of the faults deal with the ability of the macro to perform the function it was designed for. The macros are modeled so that the input test vectors test the macros as well as the network connecting them.

A few of the faults that occur within a macro are untestable without some type of dynamic test. Each D

```
  4 OR2        pin 2 stuck-at-0  remains undetected.
  6 AND3       pin 1 stuck-at-1  remains undetected.
  6 AND3       pin 2 stuck-at-1  remains undetected.
  7 AND3       pin 3 stuck-at-1  remains undetected.
 19 74         Preset stuck-at-one fault
         19 NAND3     pin 1 stuck-at-1 remains undetected.
 19 74         Clear stuck-at-one fault Cl while Ck 1
         20 NAND3     pin 2 stuck-at-1 remains undetected.
 19 74         Preset stuck-at-one fault.
         25 NAND3     pin 1 stuck-at-1 remains undetected.
 19 74         Clear stuck-at-one fault Cl while Ck 1
         26 NAND3     pin 2 stuck-at-1 remains undetected.
 20 74         Preset stuck-at-one fault
         31 NAND3     pin 1 stuck-at-1 remains undetected.
 20 74         D input stuck-at-zero fault.
         31 NAND3     pin 2 stuck-at-1 remains undetected.
 20 74         Clear stuck-at-zero fault.
         31 NAND3     pin 3 stuck-at-1 remains undetected.
 20 74         Clear stuck-at-one fault Cl while Ck 1
         32 NAND3     pin 2 stuck-at-1 remains undetected.
 20 74         Clock or Clear stuck-at-zero fault
         33 NAND3     pin 1 stuck-at-1 remains undetected.
 20 74         Clock stuck-at-one fault.
         33 NAND3     pin 2 stuck-at-1 remains undetected.
 20 74         Ability to hold clear ck high.
         34 NAND3     pin 1 stuck-at-1 remains undetected.
 20 74         Clear or Clock stuck-at-zero fault.
         35 NAND3     pin 2 stuck-at-1 remains undetected.
 20 74         Flip-flop ability to hold set
         35 NAND3     pin 3 stuck-at-1 remains undetected.
 20 74         Flip-flop ability to hold clear
         36 NAND3     pin 3 stuck-at-1 remains undetected.

The fault coverage was 64.18%.
```

Figure 11(c), Macro Mapped Undetected Fault File

flip-flop contains two faults that can only be detected by dynamic testing of the circuit with close attention to the timing considerations. These faults are undetectable under the conditions assumed by the fault grader so they have been removed initially. They do not appear in any of the fault lists and are not counted in the fault coverage. They can only appear in the detected fault list and then only if the input vectors are unrealistic.

The fault grader was designed for use with relatively small circuits. Speed was important because the fault grader will probably be used as an iterative design tool. A set of vectors would be fault graded, modified and graded again until a set of vectors achieves complete fault coverage. This process requires the repeated use of the fault grading program and would be grueling if the fault grader was excessively slow.

Because it was designed for student use, the operation of the program was kept as simple as possible. The Micro Logic 2 package, which is itself easy to use, is used as the circuit editor. Use of the fault grader in conjunction with the Micro Logic 2 package only requires a few commands once the circuit has been designed. The input vectors are in ASCII so they can be edited with whichever editor the user is most familiar with. Care was also taken to simplify the addition of elements to the libraries. Probably the hardest addition would be a macro addition because it is necessary to define the internal faults in relation to the periphery of the macro. The fault grader itself can be used to assist in this process by fault grading the expanded model of the macro as a circuit.

The program was also written to be portable. It was written on an AT&T UNIX PC but is used on an IBM PC clone. The program was written in C because of the growing acceptance of it as an engineering programming language, and because of C's reputation for portability. The bit operations available in C were also a determining factor. The element libraries are C functions so their operation is traceable.

5. Conclusions and Recommendations

The fault grader is a useful engineering tool. It supports the production of a good set of input test vectors that can be used to test a circuit. This leads to a greater confidence in the circuits that pass the test. Most of the possible physical failures are included in the single stuck-at-x fault model and if the fault grader gives the input vectors a good fault coverage figure the test when applied to the circuit will catch almost any problem.

Use of the fault grader will usually be iterative. An initial set of test vectors is prepared and evaluated with the fault grader. Then modifications of the test vectors are made, guided by the outputs from the fault grader until complete fault coverage is achieved. Outputs of the fault grader can be used to determine which of the input vectors are effective and to indicate what parts of the circuit remain untested and will require additional test vectors.

One possibility for further development of the package would be a method of generating a test vector set that could detect undetected faults. It would also be an improvement if the fault grader can be made to integrate more closely with the Micro Logic 2 package, specifically, to be able to perform a fault grading from within the Micro Logic 2 package. Another useful addition would be the ability to insert the undetected faults back into a Micro Logic 2 drawing file so they could be displayed, graphically overlaid on the circuit diagram.

Appendices

The User Manual for the fault grading program appears in appendix A. The documented C language source code for the fault grader appears in appendix B. The files are listed in alphabetic order not the order of importance. To follow the flow of the code start with the 'main' function titled <u>Fault Grader</u>. The two library files only appear in part because of their extensive length. The gates library appears in appendix C, and the macros library appears in appendix D.

Table of Contents for the Appendices.

Appendix A

Fault Grader's User Manual

by

Joseph W. Naab

May 15, 1988

1. Introduction

This is a C language software program that evaluates the ability of input test vectors to test a circuit. The program is designed to run on the outputs of another program called Micro Logic 2, which is a graphics driven digital logic simulation package. The fault grader reads the netlist files generated by the Micro Logic 2 package to form a model of the circuit. Once a model has been formed a collapsed set of faulted circuit models are constructed using the single stuck-at-x fault model. The faulted and unfaulted circuits are simulated using a event driven parallel fault simulator and outputs which explain which faulted circuit models were detected or remained undetected are produced.

The outputs of the fault grader can be used to improve the quality of the input test vectors. The undetected fault outputs show which parts of the circuit remain untested and the detected fault outputs show which input vectors do not detected any faults.

2. Starting out

If Micro Logic 2 does not reside on the system you are using it must be installed. After it is installed it will be necessary to construct an output component. Run the Micro Logic 2 package and get into the component editor, open the component window, go to the second page of components and select a blank component. The blank component should be redefined as follows:

Name	OUTPUT
Name of shape used	DELAY
Definition	NULL
Text X location	-50
Text Y location	0
Text	OUTPUT

The rest of the definitions can be ignored. Escape from the component editor and confirm that the OUTPUT component now appears in the component window on the right hand side of the screen in the page above the BUFFER element. The page above BUFFER is accessed by clicking the component window up arrow.

Once the Micro Logic 2 package has been installed and the OUTPUT component created, it is necessary to compile the source code for the fault grader unless the fault grader is already installed or has been ported from an identical system. If the system you are using does not support 32-bit 'long' integers then it is necessary to change the constant 'WORDLENGTH' in the file 'define.c' from 32 to the number of bits in a 'long' integer. The exact method of compiling differs from system to system but the object is to compile and link all the files that have the extension 'c'. These are the fault grader's source files:

```
assign.c
collapse.c
define.c
evaluate.c
gates.c
input.c
macro.c
macrofault.c
macros.c
main.c
push.c
queue.c
simulate.c
unusedmacros.c
vector.c
```

If the system you are on is a UNIX system the source code can be compiled with the command:

```
  cc *.c -o fault
```

Which will create an executable file called 'fault' which will be the fault grader if the above files are the only files in the directory that have the extension 'c'.

If you have Turbo C on your system then run Turbo C and 'make' the project file 'fault'.

3. Circuit Model for the Fault Grader

After drawing the circuit using the Micro Logic 2 package and attaching 'OUTPUT' components on all primary outputs, save the drawing file using the 'Save Drawing' option in the 'File' pop down window. Now print a netlist report to disk with the macros expanded using the 'Print' pop down window and selecting the 'Expand Macros' and the 'Disk' options, this generates the gate-level netlist of the circuit. If a macro was used in the drawing then in is necessary that the file be resaved but with the letter 'M' appended to the filename, again use the 'Save Drawing' option of the 'File' pop down window. Print another netlist report to disk but with the macros unexpanded, again using the 'Print' pop down window and select the 'Expand Macros' option again to remove the check mark so the netlist is unexpanded. This creates the macro-level netlist of the circuit which is used by the fault grader to produce macro related undetected fault outputs.

4. Input Test Vectors

The input test vectors are read from a file you must create with the same base name as the circuit but with the extension '.VEC'. This a an ASCII file that contain the input vectors, each line of the file corresponds to a input vector. The columns of the file correspond to the data elements, i.e. column 1 is DATA1. Each DATA component must have its own column filled with either a 1 or a 0. No blank lines, spaces, or comments may appear in the input vector file.

The fault grader does not support the Micro Logic 2 CLOCK component. CLOCK components are not supported by the fault grader because the Micro Logic 2 package will not produce an output file containing the clocking information. Circuits are clock by dedicating a DATA component as a clock. Two input vectors are required for each clock period by circuits that contain positive edge-triggered elements. The first vector defines the state of the inputs that are not a clock while the clock input is 0. The second vector maintains these input values and changes the clock to a 1.

5. Running the Fault Grader

The fault grader can be passed the name of the circuit on the command line or the fault grader will prompt for it. This is the only information the fault grader needs. It will read the files that have the circuit name as the base name and the extensions 'NET' and 'VEC'. If the a file ending in 'M.NET' exists it will be read also. Two or three files will be produced along with output to the screen. The output to the screen is time ordered but the files are sorted by content. The file with the circuit's base name and the extension 'DFA' will contain the detected fault output statements. The file with the extension 'UFA' will contain the undetected fault output statements, and the file with an extension of 'MFA' will contain the undetected fault statements mapped to the periphery of the macros but will only appear if the 'M.NET' file was found.

6. Outputs

The outputs of the fault grader identify the location of a fault by using the number and name of the gate or macro as it appeared in the netlist file, gates from the gate-level file and macros from the macro-level file. The pin number is also given and is defined by the order that the pins are listed in the netlist file. The detected fault outputs also contain the number of the input vector that detected the fault. The relative merit of the individual input vectors can be found using these output statements. The undetected fault files indicate which parts of the circuit remains untested. This information can be used to guide the writting of additional input vectors designed to test the parts of the circuit that remain untested.

To relate the gate and macro number to the schematic drawing produced by the Micro Logic 2 package, select the 'View' pop down window and deselect the 'Disply comp text' option and select the 'Display comp nos.' option. This will display the component numbers rather than the component's names. If macros appear in the circuit the only way to have the schematic's gate numbers agree with the netlisting's gate numbers is to have the macros' numbers greater than any gate's number. The macros do not appear in the expanded macros netlist file so gates will have the missing macros' numbers. Since Micro Logic 2 numbers sequentially in time adding the macros as the last circuit element will give them the largest numbers. Drawing the rest of the circuit without the macros is not the easiest method. Instead draw the circuit without regard to the component numbers then force the renumbering of the macros. The best way is to move each macro to a new location that will allow easy deletion of them later, as each macro is moved replace it with a new macro of the same type. Once all the macros have been replaced go through and delete the moved macros. The macros will all have numbers greater than any of the gate's numbers and the expanded netlist numbers will agree with the schematic's numbers.

The gates used for macro expansions always appear after the gates appearing in the schematic. The order of the macros is also the order of the gates used to expand them. To see the gate level model of a macro, load the drawing file with the macro's name, be careful not to modify these drawings. The gates internal to a macro are listed in the same order as they appear in the macro's drawing file.

7. Libraries

Two libraries are used by the fault grader and the elements used in the circuit model must be defined by one of the libraries. The libraries already contain all the elements that appear in the Micro Logic 2 package except for the CLOCK components. The gate library defines the properties of the individual gates and contains two C language functions that perform fault collapsing and simulation. The macro library defines the fault properties of the macros used in the circuit model. Information used to remove unused macro parts from the fault grading, map faults internal to macros to their periphery and define the number of gates used in the macro's expansion is found in the macro library. The details and concepts for adding a new gate or macro are found in the libraries' internal documentation and in more detail in a Masters Thesis written by Joseph W. Naab in May of 1988 for the Department of Electrical and Computer Engineering at Kansas State University.

Appendix B

Fault Grader's Source Code Listing

by

Joseph W. Naab

April 27, 1988

```
/*
                          Assign Function

                          Joseph W. Naab
                          April 27, 1988
                           Version 1.0

DESCRIPTION: Assign is the function that performs the
    presimulation process of determining the faulted cir-
    cuits.  It goes through the gates looking for faults,
    when it finds one it adds it to the fault simulation
    list and marks the gate as well as removing the fault
    from the gates 'state' variable.  The fault simulation
    list is a list of all the faulted circuits that are in
    the current simulation word.  It is formed using the
    structure 'fault' described in the file 'define.c'.  Ba-
    sically it holds all the information necessary to dif-
    ferentiate the faulted circuit from the unfaulted cir-
    cuit.

RETURN: The function returns the number of faults that were
    actually assigned to the fault simulation list.  If
    there were no faults left in the circuit a zero is re-
    turned.  */


#include "define.c"

static int next = 1;

int assign()
{
  extern struct FAULT parallel[];
  extern struct LIST  list[];
  extern struct GATES gates[];
  extern int  num_gates;

  int faults;

  int bit, shift;

  /* This is the part that walks through the gates.  It uses
  'next' as a static pointer to the present position so that
  each time the function is entered it already knows the
  current gate.  The gates are walked through in numeric
  order.  */
  bit = 0;
  for( ; next <= num_gates; next++ )
  {
    shift = 0;
    if( list[next].state )
      if( faults = list[next].state &
          ~( -1 << 2*( gates[list[next].offset].pins - 1)))

        /* The following section fills the fault  simulation
        list  for  each  fault.   The  first  section is for
        stuck-at-zero faults  the  second  for  stuck-at-one
        faults.   The  'andmask' and 'ormask' bit fields are
        used to mask the appropriate bit of the faulted cir-
        cuit  to  the  correct  stuck-at-value.  Only one of
        them is  necessary for each faulted circuit but they
        are  both  used anyway to reduce the decision making
        necessary for the code.  */

        while( faults )
        {
          list[next].stable = 1;
          if( faults & 1 )
          {
            parallel[ bit ].element = next;
            parallel[ bit ].pin = shift/2 + 1;
            parallel[ bit ].fault = 0;
            parallel[ bit ].andmask = -1 ^ ( 1 << bit );
            parallel[ bit ].ormask = 0;
            list[ next ].state &= -1 << shift/2 + 1;
            if( bit++ == WORDLENGTH - 1 )
              return WORDLENGTH - 1;
           }
          if( faults & 2 )
          {
            parallel[ bit ].element = next;
            parallel[ bit ].pin = shift/2 + 1;
            parallel[ bit ].fault = 1;
            parallel[ bit ].andmask = -1;
            parallel[ bit ].ormask = 1 << bit;
            list[ next ].state &= -2 << shift/2 + 1;
            if( bit++ == WORDLENGTH - 1 )
              return WORDLENGTH - 1;
          }
          shift +=2;
          faults >>= 2;
        }
  }
  return bit;
}
```

```
/*
                          Collapse Function


                          Joseph W. Naab
                          April 27, 1988
                           Version 1.0


DESCRIPTION:  Collapse is the function responsible for
    creating a collapsed set of faults.  It goes through the
    following steps.

      - set all fanouts to stuck-at-x faults.
      - set all primary inputs to stuck-at-x faults.
      - push the faults through the circuit.

    To do the first step it goes through the netlist looking
    for nets with fanouts greater than one.  When it finds
    one it sets all of the inputs tied to that net to
    stuck-at-x faults.  The second step uses a list of data
    element, which are the primary inputs, to assign an
    stuck-at-x fault to an input tied to each date element.
    Fanouts are not considered because they would already
    have stuck-at-x faults from the proceeding step.  The
    third step is done by traversing the gate list looking
    for a gate that is still unstable.  A gate becomes un-
    stable whenever a fault is assigned to it and stable
    after it has had its faults 'pushed'.  The faults will
    be completely collapsed when there exist no unstable
    gates in the circuit.


FUNCTIONS CALLED: The only function used by this routine is
    the function 'push' which takes care of the unstable
    gates by performing fault rules of the gate.  Push is
    described in more detail in 'push.c'.  */


#include "define.c"

int collapse()

{
  int i, j;
  extern struct NET      net[];
  extern struct GATES    gates[];
  extern struct LIST     list[];
  extern struct LIST     *data[];
```

```
    extern int              num_data;
    extern int              num_gates;

    /* First go through the net list and find  the  nets  with
    fanouts  greater  than  one  and assign the inputs tied to
    those nets stuck-at-x faults.  */

    i = 0;
    while( net[ ++i ].driver )
      if( net[ i ].fanout > 1 )
        for( j=net[i].fanout; j--; )
        {
          list[ net[ i ].elements[ j ] ].stable = 1;
          list[ net[ i ].elements[ j ] ].state |=
                  ( (long)3 << ( 2 * (net[i].pin[j]-1)));
        }

    /* Second go through the data element list an  assign  one
    of the inputs tied to each a stuck-at-x fault. Fanouts can
    be ignored because they have been taken  care  of  by  the
    previous section.  */

    for( i =0; i <= num_data; i++ )
      {
        list[ net[ data[ i ]->net].elements[ 0 ] ].stable = 1;
        list[ net[ data[ i ]->net].elements[ 0 ] ].state |=
           ((long)3 << ( 2 * (net[data[i]->net].pin[0]-1)));
      }

    /* Go through the gate list looking  for  unstable  gates,
    gates  that have newly assigned faults.  For each unstable
    gate found pass it to the  function  'push'.  Repeat  the
    this  process  until  there are no more unstable gates, go
    all the way through the gate list without calling  'push'.
    */

    j = -1;
    for( i=0; i != j; i = ( i % num_gates ) + 1 )
      if( list[ i ].stable )
      {
#       ifdef DEBUG
          printf( "collapse: pushing element %d \n", i );
#       endif
        j = i;
        push( i );
      }
}
```

```
/*
                         Define file

                      Joseph W. Naab
                      April 20, 1988
                       Version 1.0


    DESCRIPTION:
        This file is used by every function to define the
        structures that they work with. The structures have
        individual descriptive documentation as well as in
        line documentation. All of the structures used in
        this program are used by more than one function so
        they are defined here because the compiler must be
        able to see the structure definition.

        Several of the structures in this file contain an in-
        teger that is used as an index into another struc-
        ture. This was done to make the fault grader some-
        what memory efficient. The differing size and char-
        acter of the individual elements makes it difficult
        or wasteful to use a structure large enough to hold
        the worst case. So another structure was defined and
        just an index to and the number of entries in the new
        structure can be stored. An example is the integer
        'pins' in the structure 'list'. The number of pins
        is variable so the nets tied to each pin are kept in
        a integer array. The index to a gate's nets is held
        by 'pins'.

        The first few lines define compiler constants.

        DEBUG if defined causes diagnostic messages to be
        printed out by each function as they run. The print
        statements that are produced are crude but were put
        in during the writing of the program for debugging
        purposes. They remain for future programmers, if DE-
        BUG is undefined the lines between 'ifdef DEBUG' and
        'endif' are ignored by the compiler.

        MAXDATA defines the maximum number of data inputs
        that can appear in the circuit model.

        MAXINPUTS defines the maximum number of input vectors
        that can be used.

        MAXNETS defines the number of nets that may appear in
        the circuit model.

        MAXPARTS defines the number of gates that may  appear
        in the expanded circuit model.

        PINCOLUMN is the first column that the nets that  are
        tied to the pins of the gates may appear. */


#include <stdio.h>
/*#define DEBUG*/

#define MAXDATA    32    /* the number of data inputs        */
#define MAXINPUTS  100   /* the number of input vectors      */
#define MAXMACROS  100   /* the number of macros in model    */
#define MAXNETS    100   /* the max num of networks          */
#define MAXPARTS   100   /* max num of elements              */
#define PINCOLUMN  17    /* column pin net's list start      */
#define WORDLENGTH 32    /* the length in bits of a long     */

        /* FAULT is the structure that is used for the  fault
        simulation  list.   Each  member  of an array of type
        structure 'fault' is a faulted circuit that is  being
        currently  simulated.  The integer 'element' contains
        the number of the element where the fault is located.
        The pin of the element is stored in 'pin'.  The fault
        type, one for stuck-at-one, zero for stuck-at-zero is
        in  fault.  The next two integers, 'andmask' and 'or-
        mask', are bit field that are used during  simulation
        to  force  the  faulted  circuits  value  that of the
        fault. */

struct FAULT
{
  int    element;   /* element's index into list & number  */
  int    pin;       /* pin that fault is on                */
  long   fault;     /* the type of fault                   */
  long   andmask;   /* the AND mask for stuck-at value     */
  long   ormask;    /* the OR mask for stuck-at value      */
};

        /* GATES is the structure that is used  in  the  gate
        library.   It  is  initialized in the file 'gates.c'.
        The string pointed to by 'name' is used in  a  string
        comparison  with the element names that appear in the
        circuit model.  The integer pins tells how many  pins
        are  associated  with  the gate.  The integer time is
        used during simulation to determine  the  gate  delay
        for  the event driven simulator. The variable 'fault'
        is a pointer to an  user  supplied  integer  function
        that  will  be used during the fault collapsing stage
        of the fault grading to determine the fault state  of
        the gate. The variable 'simulate' is a pointer to the
        user supplied integer function that is used to  simu-
        late  the  gate.  It is called whenever a gate of the
        type 'name' is simulated. */

struct GATES
{
  char   *name;            /* ascii name of the element      */
  int    pins;             /* element's number of pins       */
  int    time;             /* element's simulation delay time*/
  int    (*fault)();       /* pointer to fault compress funct*/
  int    (*simulate)();    /* pointer to element sim function*/
};

        /* LIST is the structure used to hold the gates  that
        define  the  circuit  model  internal  to  the  fault
        grader.  Each gate that appears in the circuit's  ex-
        panded  netlist  file creates a member of this struc-
        ture.  The integer offset is an index  into  the  li-
        brary  of gates.  It is assigned the value of the in-
        dex to the structure 'gates' member whos name matched
        the  name in the netlist file.  The integer  'stable'
        is used in both the fault compression and  simulation
        processes.   It  is basically a flag variable, in the
        fault compression it is set if the gate must have its
        faults  pushed  and  in the simulation it is set if a
        faulted circuit has a fault on this  gate.   The  in-
        teger  'state'  is  really  a thirty-two bit long bit
        field.  It is used by the entire fault grader to keep
        track  of the faults for this gate.  The least signi-
        ficant two bits represent both of the stuck-at faults
        on  the  first  input, the next two bits are used for
        the second input and so on.  The integer  pins  is  a
        index  in  to the integer array 'parts' that contains
        the number of the net that is tied to each of the in-
        put  pins.   The integer net is the number of the net
        that is driven by the output of this gate. */

struct LIST
{
  int    offset;           /* the index into GATES library  */
  int    stable;           /* fault flag                    */
  long   state;            /* hold's fault information      */
  int    *pins;            /* pointer into array input nets */
  int    net;              /* net driven by this element    */
};
```

```
        /* MACROLIST is the structure that holds  the  macros
        that  appear in the circuit model.  'offset' is a in-
        dex into the macro library  and  and  number  is  the
        number of the line the macro appeared on in the unex-
        panded netlist file. */

struct MACROLIST
{
  int    offset;
  int    number;
};

        /* MACROS is the structure used by the structure  li-
        brary.   It is initialized in the file 'macros'.  The
        string pointed to by 'name' is use in a  string  com-
        parison  with  the  elment  names  that appear in the
        unexpanded circuit model.  The integer 'gates'  holds
        the  number of gates that are used to expand the mac-
        ro.  It is used to enable the fault grader to  calcu-
        late  the  boundaries  of  the macros in the expanded
        circuit model.  The integer  'list'  holds  an  index
        into the structure 'macrofaults' where the faults for
        the macro are defined in terms of the  macro's  peri-
        phery.   The  integer  'parts'  is  an index into the
        structure 'macroparts' which  holds  the  information
        necessary to break compound macros up into their sub-
        functions. This is used to remove unused parts of the
        macro  from  the fault graders consideration. It will
        be zero if the macro is a single  functional  entity.
        */

struct MACROS
{
  char   *name;            /* The name of the macro          */
  int    gates;            /* #gates in expanded macro       */
  int    faults;           /* #faults defined by each macro  */
  int    list;             /* list of faults for this macro  */
  int    parts;            /* to 'parts' of compound macros  */
};

        /* MACROFAULTS is the structure that  is  used  as  a
        secondary  storage space in the 'macros' library.  It
        identifies a specific fault as  an  offset  from  the
        start  of  the  macro  and contains the corresponding
        mapping statement which is output to  give  the  user
        the  relationship  between  the internal fault and an
        external fault, if any.  The integer  'gate'  is  the
        number  of  gates  offset  from the first gate in the
        macro the gate with the fault on it appears.  The in-
        teger  'pin'  is the pin on that gate where the fault
        appears.  The integer 'fault' is the type  of  fault,
        one  for  a  stuck-at-one,  zero for a stuck-at-zero.
        The string 'map' hold the statement that will be out-
        put  along  with  the  undetected stuck-at statement.
        The string should relate  the  internal  fault  to  a
        fault on the periphery of the macro.  If the internal
        fault is undetectable the string should  be  initial-
        ized to NULL, this will cause the fault grader to ig-
        nore the existence of that fault. No undetected fault
        statement will be generated for it and it will not be
        included in the fault coverage calculations. */

struct MACROFAULTS
{
  int    gate;             /* the gate the fault appears on  */
  int    pin;              /* the pin the fault appears on   */
  int    fault;            /* the type of fault that detected*/
  char   *map;             /* the output for this fault      */
};

        /* MACROPARTS is a structure that is used as  a  sub-
        structure  in  the  library 'macros'.  It each member
        describes a subfunction of a macro that has more than
        one function, e.g. a dual D flip-flop.  The parts are
        identified so that any unused  part  can  be  removed
        from  the  fault  grading.  The integer 'gate' is the
        number of gates offset from the  first  gate  of  the
        macro.  The gate is of interest because a unconnected
        input on this gate means this part of  the  macro  is
        unused.  The integer 'pin' is the pin that is a macro
        input.  If the pin is unconnected then that  part  of
        the  macro  is  unused.  The integer 'length' is the
        number of gates used to expand that part of the  mac-
        ro. */

struct MACROPARTS
{
  int    gate;             /* the offset to macro input      */
  int    pin;              /* #pin has zero net if unused    */
  int    length;           /* #gates to expand macro part    */
};

        /* NET is the structure that hold the net's  informa-
        tion  in the internal model of the circuit.  The nets
        are were the  simulation  values  are  stored  during
        simulation.  A nets logical value is determined sole-
        ly by the gate's output that it is tied to.  The  in-
        teger  'fanout' contains the number of inputs tied to
        the net.  The integer 'driver' contains the number of
        the gate that drives the net.  The integer 'logic' is
        a bit field used as a simulation field, each  of  the
        bits  in  'logic'  is  used  for  a different circuit
        model.  The integer 'defined' is also a bit field and
        is  used  to signify whether or not the logical value
        in 'logic' is a defined logical value. A set  bit  in
        'defined'  means that the corresponding logical value
        in 'logical' is the known logical value of  the  net.
        The integer array 'elements' provides space for stor-
        ing the number of the elements that have  input  pins
        tied  to  the  net.  The integer array 'pin' give the
        pin number that is tied to the net of the correspond-
        ing elment in 'elements'. */

struct NET
{
  int    fanout;          /* #inputs tied to the net        */
  int    driver;          /* #gate driving the net          */
  long   logic;           /* value of the net during simul. */
  long   defined;         /* defined state during simul.    */
  int    elements[10];    /* elements tied to the net       */
  int    pin[10];         /* pins of elements tied on net   */
};

        /* QUEUE is the structure used to queue the  elements
        for  future simulation by the event driven simulator.
        The queue is in the form of a linked list  so  'next-
        time'  is a pointer to the next queue structure.  The
        integer 'time' is the time that the elements  in  the
        current  structure  are  scheduled  for.  One  queue
        structure is used for  each  time,  several  elements
        could  be  scheduled  at  the same time in which case
        they would use the same queue structure member.   The
        character  array  'element'  a  bit field of variable
        length.  If more then sixteen elments are in the cir-
        cuit model the queue is allocated extra memory at run
        time which is appended to  'element'.   Each  bit  in
        'element'  corresponds to an element in the model, if
        the a bit is set then the  corresponding  element  is
        scheduled. */

struct QUEUE
{
  struct QUEUE *nexttime;/* pointer to the next time        */
  int    time;           /* the queue element's time        */
  char   element[1];     /* list elments in queue element   */
};
```

```
/*
                          Evaluate Function

                            Joseph W. Naab
                            April 27, 1988
                             Version 1.0


DESCRIPTION:  This function is used by the 'simulate' func-
    tion to perform the simulation of the individual gates.
    The actual simulation is done by the simulation function
    that appears in the gate library.  This function sets up
    the input variables and calls the library simulation
    functions.


    Each library function is passed the same information, two
    arrays that contain the logical and defined states of the
    gate's inputs.  This function fills the array from the
    state and defined information that is stored by the nets.
    The state and defined states of each input pin is read
    from the net that the input pin is tied to and the infor-
    mation is stored in the array that is passed to the simu-
    lation function.


    This is also the function that checks the gates to see if
    a faulted circuit is associated with it.  If there is one
    the bit associated with the faulted function is forced to
    the fault state by masking with the corresponding masks
    that are already stored in the in the fault simulation
    list, 'parallel'.


    After calling the library simulation function the new
    output of the gate is compared to the old output of the
    gate.  If a change has occurred the net driven by the
    current gate is updated to the new state and the time the
    net changed its value is calculated.  The funtion returns
    a 1 if the output changes and a 0 if the output remains
    unmodified.


ARGUMENTS: 'element' is the number of the element that is to
    be simulated.

    'time' is the current simulation time and is passed back
    as the time at which the net's state changes.
```

```
RETURN: 1 if 'element' outputs has changed and a 0 if  it  is
    unchanged.


FUNCTIONS CALLED: Calls the simulation functions in the  gate
    library  through  the  use  of  function  pointers.   The
    pointers appear in the library structure 'GATES'.  */



#include "define.c"

int evaluate( element, time )

  int element,
      *time;
{
  extern struct NET   net[];      /* holds net info         */
  extern struct GATES gates[];    /* the gate library       */
  extern struct FAULT parallel[]; /* fault simulation list*/
  extern struct LIST  list[];     /* holds element info     */

  extern long   linputs[];  /* logical input states         */
  extern long   dinputs[];  /* defined input states         */

  int           i;
  long          loutput;    /* logical output state         */
  long          doutput;    /* output defined state         */

    /* Fill the arrays that contain the logical  and  defined
    states of the inputs by looking at the states of the nets
    that are tied to the input pins.*/

  for( i = gates[ list[element].offset ].pins-1; i--; )
  {
    linputs[ i ] = net[ *(list[element].pins + i ) ].logic;
    dinputs[ i ] = net[ *(list[element].pins + i ) ].defined;
#   ifdef DEBUG
      printf( "eval: gate %d pin %d logic %x defined %x\n",
               element, i+1, linputs[ i ], dinputs[ i ]);
#   endif
  }

    /* Check the gate for the presence of a  faulted  circuit
    model.   If  one exists then go through the fault simula-
    tion list to find the associated fault and mask the input
    bits to the stuck-at value. */

  if(  list[element].stable )
```

```
    for( i = 0; i < WORDLENGTH - 1 ; i++ )
      if( parallel[ i ].element == element )
      {
        linputs[ parallel[ i ].pin - 1 ] &= parallel[i].andmask;
        linputs[ parallel[ i ].pin - 1 ] |= parallel[i].ormask;
      }

    /* Call the simulation function and see  if  the  outputs
    returned  have  changed.   If they have update the driven
    net's state  and  calculate  the  time  the  net  changes
    states.   Return a 1 if the output has changed and a 0 if
    it remain unmodified.*/

  (*( gates[ list[ element ].offset ].simulate ))
      ( linputs, dinputs, &loutput, &doutput );

  if((( net[ list[ element ].net ].logic ^ loutput ) |
        ~net[ list[ element ].net ].defined | ~doutput)
        & ( net[ list[ element ].net ].defined | doutput ))
  {
#   ifdef DEBUG
      printf( " eval: change in element %d, new logic %x,",
                element, loutput);
      printf( " new define %x, old logic %x old define %x\n",
                doutput, net[ list[ element ].net ].logic,
               net[ list[ element ].net ].defined );
#   endif
    net[ list[ element ].net ].logic = loutput;
    net[ list[ element ].net ].defined = doutput;
    *time += gates[ list[element].offset ].time;
    return 1;
  }

  return 0;
}
```

```
/*
                              Fault Grader

                             Joseph W. Naab
                             April 27, 1988
                              Version 1.0


DESCRIPTION: This is the main routine for the fault grading
    program. It contains the declaration of the global vari-
    ables, the file opening code, and the calls to the pri-
    mary subfunctions. The basic order of operation is:

      read in the model of the circuit
      read in the macro model of the circuit if it exists
      produce a collapsed set of faulted circuits
      remove unused macro parts
      read in the input vectors
      repeat until all faulted circuits are simulated
        assign a set of faults
        simulate the assigned fault set


    The model of the circuit is read from the file
    'filename.NET'. If the model input is unsuccessful the
    program aborts. An attempt is made to open the file
    'filenameM.NET', if it is opened successfully the macro
    level model of the circuit is read from that file. A
    collapsed set of faulted circuits are constructed for the
    circuit model. If a macro file was read then the unused
    parts, if any exist, of macros are removed from the set
    of faulted circuits. The file 'filename.VEC' is opened
    and the input vectors are read in. The first set of
    faulted circuits are put on the fault simulation list and
    they are simulated. The next set of faulted circuits are
    put on the list and simulated until there are no faulted
    circuits left unsimulated.


ARGUMENTS: The base filename of the circuit's files can be an
    argument on the command line.


RETURN: Three output files are produced as well as outputs to
    the screen. The three files all have the base filename
    but with the extensions, 'DFA', 'UFA', and 'MFA' which
    are the detected fault file, the undetected fault file,
    and the macro fault file, respectively.
```

```
FUNCTIONS CALLED: input(),  macro(),  collapse(),  unusedmac-
    ros(),  input_vectors(),  assign(),  and simulate().  The
    function 'input' reads  the  gate-level  circuit  netlist
    file  'filename.NET'  and is responsible for creating the
    internal model of the circuit.  The function  'macro'  is
    called  if  the file 'filenameM.NET' opens correctly.  It
    is the file that read the macro-level model of  the  cir-
    cuit  and  fills  the 'macrolist' with the macros that it
    finds.  The function 'collapse' creates a  collapsed  set
    of  faulted  circuits  using  the gate-level model of the
    circuit created by 'input'.  The function  'unusedmacros'
    is only called if the macro-level file was read.  It goes
    through the 'macrolist' looking for unused parts of  mac-
    ros,  if any are found the faulted circuit models associ-
    ated with the unused macro parts  are  removed  from  the
    collapsed  set  of  faults.  The function 'input_vectors'
    reads the file 'filename.VEC' for the input vectors which
    appear there in ASCII.  The functions 'assign' and 'simu-
    late' are called repeatedly until all of the faulted cir-
    cuits  have  been  simulated.  'assign' creates the fault
    simulation list for the parallel fault simulator.  'simu-
    late'  simulates  the  unfaulted  and faulted circuits in
    parallel generating the output of the fault grader.  Out-
    puts  for  detected faults are produced as the faults are
    detected.  Undetected and macro fault  outputs  are  gen-
    erate  after the simulator has simulated all of the input
    vectors.  */

#include "define.c"

struct NET        net[100] = { 0 };
struct LIST      *data[ 32 ];
struct LIST      *output[ 32 ];
struct LIST       list[ MAXPARTS ] = { 0 };
struct MACROLIST macrolist[ MAXMACROS ];
struct FAULT      parallel[ WORDLENGTH ];

long           dinputs[16];
int            expanded = 32000;
long           linputs[16];
int            num_vectors;
int            num_output = -1;
int            num_data = -1;
int            num_gates;
int            num_macros = 0;
int            part[ 6*MAXPARTS ] = { 0 };
int            vector[ 32 ][ MAXINPUTS ];

main( arg1, arg2 )
```

```
  int       arg1;
  char      **arg2;

{
  extern struct GATES gates[];

  int       assign();          /* assigns bits to faults       */
  int       collapse();        /* collapses the fault set      */
  int       input();           /* reads in the circuit model   */
  int       input_vectors();   /* reads ASCII input vectors.   */
  int       macro();           /* reads macro circuit model    */

  int       faults,            /* #faults being simulated      */
            totalfaults = 0,   /* # of fault in collapsed set  */
            detectedfaults=0,/* # of detected faults         */
            i,j;

  char      filename[40];      /* name of I/O files            */
  char      circuit[40];       /* generic circuit name, "arg2'*/

  FILE      *file,             /* input files pointer          */
            *macrofile,        /* circuit.MFA file, macro out  */
            *detected,         /* circuit.DFA file, detect out*/
            *undetected;       /* circuit.UFA file, undet out  */

    /* Open the gate-level circuit model file  'filename.NET'
    and  input the circuit model.  If and error occurs during
    input the function 'input' will return a -1 and the  pro-
    gram will terminate. */

  if( arg1<2 )
  { printf( "Enter the name of the file containing the " );
    printf( "circuit model. \n");
    scanf( "%s", circuit );
  }
  else
    strcpy( circuit, arg2[1] );

  strcpy( filename, circuit );
  strcat( filename, ".NET" );
  file = fopen( filename, "r" );
  while( !file )
  {
    printf( " Unable to open %s correctly,", filename );
    printf( " reenter file name.\n" );
    scanf( "%s", filename );
    strcpy( circuit, arg2[1] );
    strcat( filename, ".NET" );
    file = fopen( filename, "r" );
  }

    /* Open the input vector file 'filename.VEC' and read  in
    the input vectors. */

  if( input( file ) )
  {
#   ifdef DEBUG
      for( i=1; i<=num_gates ; i++)
      {
        printf( "main: net %d fanout %d", i, net[i].fanout,
                " driving element %d\n", net[i].driver );

        for( j=net[i].fanout; j--;)
          printf( " element %d pin %d",net[i].elements[j],
                  net[i].pin[j] );

        printf( "\nmain: element %d inputs ", i );

        for( j=0; j < gates[ list[i].offset ].pins; j++ )
          printf( " pin %d on net %d,",j, *(list[i].pins+j));
        printf( "\n");
      }
#   endif

    /* Try to open the macro-level file  'filenameM.NET',  if
    opens successfully call 'macro' to read the file and also
    open the output file 'macrofile'.*/

    strcpy( filename, circuit );
    strcat( filename, "M.NET" );

    if( file = fopen( filename, "r" ))
    {
#   ifdef DEBUG
      printf( "main: reading in macro file \n" );
#   endif
      macro( file );

      strcpy( filename, circuit );
      strcat( filename, ".MFA" );
      macrofile = fopen( filename, "w" );

    }

    /* Create a collapsed set of faulted circuit models. */

    collapse();
    /* If the macro-level file was  read  remove  the  unused
    parts of the macros from the fault simulation. */

    if( file )
      unusedmacros();

#   ifdef DEBUG
      for( i =1; i <= num_gates; i++)
      {
        printf( "main: element %d has faults %x \n",
                   i, list[i].state );
      }
#   endif

    strcpy( filename, circuit );
    strcat( filename, ".VEC" );
    file = fopen( filename, "r" );
    while( !file )
    { printf( " Unable to open %s correctly,", filename,
               " enter input vector file name. \n" );
      scanf( "%s", filename );
      file = fopen( filename, "r" );
    }

    num_vectors = input_vector( file );

    strcpy( filename, circuit );
    strcat( filename, ".DFA" );
    detected = fopen( filename, "w" );

    strcpy( filename, circuit );
    strcat( filename, ".UFA" );
    undetected = fopen( filename, "w" );

    /* Create a fault simulation list by  assigning  some  of
    the  faulted circuits to it.  The integer returned by the
    function 'assign' is the number of  faulted  circuits  it
    placed  on  the  fault  simulation list.  While there are
    faulted circuits in the fault  simulation  list  simulate
    them and generate the fault grader's outputs. */

    while( faults = assign())
    {
#   ifdef DEBUG
      printf( "main: after assign: % assigned faults \n", i);
      for( j =0; j < faults; j++)
      {
        printf( "fault %d - gate %d pin %d  fault %d ", j,
                   parallel[j].element, parallel[j].pin,
                    parallel[j].fault);

          printf( " andmask %x ormask %x stable %d\n",
                    parallel[j].andmask, parallel[j].ormask,
                    list[parallel[j].element].stable);
        }
#   endif

    totalfaults += faults;

    detectedfaults += simulate( detected, undetected,
                                    macrofile, faults );

    }

    printf( "\n0he fault coverage was %4.2f%%.0,
             100 * detectedfaults / (float)totalfaults);

    fprintf( detected, "\nThe fault coverage was %4.2f%%.0,
             100 * detectedfaults / (float)totalfaults);

    fprintf(undetected,"\nThe fault coverage was %4.2f%%.0,
             100 * detectedfaults / (float)totalfaults);

    if( macrofile )
     fprintf(macrofile,"\nThe fault coverage was %4.2f%%.0,
              100 * detectedfaults / (float)totalfaults);
  }
}
```

```
/*
                              Input Function


                              Joseph W. Naab
                              April 27, 1988
                                Version 1.0


DESCRIPTION:  This function is responsible for  inputing  the
    circuit model.  It is the function that fills the element
    and net structures which forms the model of the  circuit.
    The  input file pointer is passed to the function and the
    function assumes that the input file is  the  gate  level
    output  file  of the Micro Logic 2 package or of the same
    format.  Very little error checking is  done  during  the
    model  input  because  it was designed to read the output
    file of another program.

    The circuit model is read in line by line.  Each line de-
    fines  a  gate in the circuit model first the gate number
    and name are read off the line.  The gate name  is  found
    in the gate library and the offset to the type of gate is
    stored in 'list' indexed by the gate number.   The  input
    net  numbers  are  read  next  and stored in both the net
    structure 'net' and a list pointed to by the gate  struc-
    ture 'list' called 'part'.  If the element is a 'data' or
    an 'output' element it receives special handling.

ARGUMENTS: 'file' is a pointer  to  the  gate  level  circuit
    model file.

RETURN: 0 if an error, 1 otherwise.

FUNCTIONS CALLED: none.  */


#include <ctype.h>
#include "define.c"

int input( file )

  FILE *file;

{
    /* A more complete description of the  external  variable
    can  be  found  in the file 'define.c' for the structures
    and 'main.c' for the integers. */
```

```
extern struct GATES   gates[];   /* library of gates      */
extern struct LIST    list[];    /* holds gate's info     */
extern struct LIST    *data[];   /* list of data inputs   */
extern struct LIST    *output[]; /* list primary outputs */
extern struct NET     net[];     /* holds netlist's info */

extern int        gate_library;  /* # gates in library    */
extern int        num_data;      /* # data in model       */
extern int        num_gates;     /* # gates in model      */
extern int        num_output;    /* # outputs in model    */
extern int        part[];        /* list of nets on input*/

int  next;          /* The next column to read input line */
char inbuf[80];     /* The input line buffer.             */
char element[15];   /* The gates name as read off input   */
int  i,j,pin,       /* Miscellaneous variables            */
     status = 1,    /* Value return by function fgets()   */
     step = 0,      /* index into 'parts' array           */
     index,         /* offset into gate library           */
     gatenum;       /* current gate number                */

  /* Look for the first line of the circuit model's defini-
  tion  signified  by  a  one in either the first or second
  space on the line. */

for( ; ( status = (int)fgets( inbuf, 80, file )) &&
        inbuf[0] != '1' && inbuf[1] != '1' ; );

  /* Continue reading in lines until the  EOF  or  a  blank
  line  is reached. Everything inside this loop is for each
  line of input.*/

while( status )
{

#   ifdef DEBUG
      printf( " model input: %s \n", inbuf );
#   endif
    sscanf( inbuf, "%d%s", &gatenum, element );

    /* Find the name of the gate in library structure 'gates'
    and  remember the offset into the library to identify the
    gate.*/

    for( i =0; strcmp(element, gates[i].name )
               && i < gate_library; i++ );

    if( i < gate_library )
      index = i;
```

```
else
{
  printf( " The circuit model contains a element ");
  printf( "unknown to the simulator %s. ", element );
  printf( " PROGRAM ABORTED !! ");

  return 0;
}

/* Set up a pointer into the  list  that  holds  the  net
numbers  of  the  nets that are tied to the input pins of
this gate.  This is done so gates with different  numbers
of input pins could be handled the same way.*/

list[ gatenum ].pins = &part[ step ];
list[ gatenum ].offset = index;

/* Pull off the net numbers of the nets that are tied  to
the input pins.  Store the net number for each pin in the
input pin net list 'parts and add the gate to  the  net's
fanout.   The  exception is the 'output' element which is
not put in the net's fanout for simulation purposes. */

next = PINCOLUMN;
if( !( strncmp( element, "OUTPUT", 6 )))
{
  output[ ++num_output ] = &list[ gatenum ];
  sscanf( &inbuf[ next ], "%d", &pin );
  part[ step++ ] = pin;
}
else
{
  j=1;
  for( i= gates[ index ].pins - 1; i; i-- )
  {

    for( ; isspace( inbuf[ next ]); next++ )
      if( inbuf[ next ]  == '\n')
      {
        printf( "ERROR: Element %d ", gatenum);
        printf( "has too few input pins\n" );

        return 0;
      }

    pin = atoi( &inbuf[ next ] );

    for( ; !isspace( inbuf[ next ]); next++ );
```

```
            if( pin )
            {
              net[ pin ].elements[ net[ pin ].fanout ] = gatenum;
              net[ pin ].pin[ net [ pin ].fanout++ ] = j++;
            }

            part[ step++ ] = pin;
          }

      /* Pull off the output net of this gate.  The net tied to
      the  gates  output  is  driven by that gate and is stored
      separately in both gate structure and the net  structure.
      */

        for( ; isspace( inbuf[ next ]); next++ )
          if( inbuf[ next ]  == '\n')
          {
            printf( "ERROR: Element %d ", gatenum);
            printf( "doesn't have output pin\n" );
          }

        pin = atoi( &inbuf[ next ] );
        list[ gatenum ].net = pin;
        net[ pin ].driver = gatenum;

        if( !( strncmp( element, "DATA", 4 )))
          data[ ++num_data ] = &list[ gatenum ];
      }

    status = ( ( int )fgets( inbuf, 80, file )
                && inbuf[0] != '\n' );
    }

    num_gates = gatenum;

      /* Go through the entire net structure  and  convert  the
      fanout  elements from the number of the net driven by the
      gate to the actually number of the gate the net  is  tied
      to. */

    for( i = 0; net[ i ].driver; i++  )
      for( j = net[ i ].fanout + 1; --j; )
        net[i].elements[j] = net[ net[i].elements[j]].driver;

    return 1;
  }
```

```
/*
                          Macro Function

                          Joseph W. Naab
                          April 27, 1988
                            Version 1.0

DESCRIPTION:  This is the function that reads  in  the  macro
    level network listing file generated by the Micro Logic 2
    package.  The function reads in the file line by line and
    compares  the  element  names with the names in the macro
    library.  When it finds and element in the macro  library
    it records the offset to the macro and the macros element
    number in the macro list.  By counting the  number  gates
    used  in the expansion of each macro, information that is
    in the macro library, the function  can  calculate  which
    gates in the circuit model are expansions of macros.  The
    Micro Logic 2 package lists all discrete gates  before it
    lists  the  gates  in the macro expansion so by known the
    total number of gates used for all  the  macro  expansion
    the  gates  used  in a macro expansion can be identified.
    This information is used in the generation of  undetected
    fault messages.


ARGUMENTS: The pointer to the file 'filenameM.NET', the macro
    netlist file, is passed to the function.


RETURN: none.


FUNCTIONS CALLED: none.  */

#include "define.c"

int macro( file )

  FILE *file;

{
  extern struct MACROS macros[];
  extern struct MACROLIST macrolist[];
  extern int num_gates;
  extern int num_macros;
  extern int macro_library;
  extern int expanded;
  char inbuf[80];
  char element[15];
  int  i;
  int  macrogates = 0;
  int status = 1;
  int newnet;

  for( ; ( status = (int)fgets( inbuf, 80, file )) &&
         inbuf[0] != '1' && inbuf[1] != '1' ; );

  while( status )
  {
#   ifdef DEBUG
      printf( " model input: %s \n", inbuf );
#   endif
    sscanf( inbuf, "%d%s", &newnet, element );

    for( i =0; strcmp(element, macros[i].name ) &&
         i < macro_library; i++ );

    if( i < macro_library )
    {
      macrolist[ num_macros ].offset = i;
      macrolist[ num_macros++ ].number = newnet;
      macrogates += macros[ i ].gates;
    }
  status = (int)fgets( inbuf, 80, file ) && inbuf[0] != '\n';
  }

  expanded = num_gates - macrogates;
}
```

```
/*
                         Macrofault Function

                           Joseph W. Naab
                           April 27, 1988
                             Version 1.0


DESCRIPTION:  This function finds the message  in  the  macro
    library  that correlate the faults internal to a macro to
    conditions on the periphery of the macro.   The  function
    is  called  by  the function 'simulate' when a undetected
    fault occurs at a gate that is internal to a  macro.   It
    finds  the fault and returns a pointer to the correspond-
    ing message.

    The fact that the gates are simulated  in  numeric  order
    and that undetected faults are also found in numeric ord-
    er.  The function walks through the  possible  faults  in
    the  library  until it finds the current fault.  There is
    no provision for backing up, the  function  always  looks
    forward through the faults associated with a macro.


ARGUMENTS:  The exact fault is passed to the function and  is
    identified  by  the  gate  and pin it appeared on and the
    type of fault.  The function returns the  number  of  the
    macro that the fault was internal to.


RETURN:  The function returns a pointer to  the  text  string
    that  maps  the  fault internal to the macro to the peri-
    phery of the macro.


FUNCTIONS CALLED: none.  */



#include "define.c"

static int current = 0,
           first,
           bottom,
           last;

char *macrofault( gate, pin, fault, macro )
```

```
  int gate,
      pin,
      fault,
      *macro;

{
  extern int    expanded;

  extern struct MACROS      macros[];
  extern struct MACROFAULTS macrofaults[];
  extern struct MACROLIST   macrolist[];

# ifdef DEBUG
    printf( "macft:look fault %d gate %d pin %d macro %d\n",
             fault, gate, pin, current );
# endif

    /* Find the macro that the gate is internal to.  The mac-
    ro is found by keeping a running sum of the gates used in
    the other macros.  When a macro  is  found  that  extends
    past  the current gate it is the macro that contains that
    gate.*/

  for( gate -= expanded; gate > last ; current++ )
  {
    last  += macros[ macrolist[ current ].offset ].gates;
    first  = macros[ macrolist[ current ].offset ].list;
    bottom = macros[ macrolist[ current ].offset ].faults +
             first;
  }

    /* Convert the gate number to its number with respect  to
    the  start  of  the macro's expansion.  Then look through
    the possible faults for the current until  a  fault  with
    the same gate number, pin number and fault type is found.
    */

  gate -= last - macros[ macrolist[current-1].offset ].gates;
# ifdef DEBUG
    printf( "macft:look fault %d gate %d pin %d macro %d\n",
             fault, gate, pin, current );
# endif
  while( first <= bottom )
  {
    if( macrofaults[ first ].gate != gate )
      first++;
    else if( macrofaults[ first ].pin != pin )
      first++;
    else if( macrofaults[ first ].fault != fault )
```

```
        first++;
      else
      {
        *macro = current-1;
        return macrofaults[ first ].map;
      }
    }
    printf( "ERROR: Fault not found in macro library.\n" );
    printf( "\tMacro %d, gate %d , pin %d, stuck-at-%d.0,
            current, gate, pin, fault );
    return "";
}
```

```
/*
                            Push Function


                           Joseph W. Naab
                           April 27, 1988
                            Version 1.0


DESCRIPTION:  This function is responsible  for  calling  the
    fault collapsing functions found in the gate  library  to
    propagate changed fault states forward through  the  net-
    list.  The function marks  the  gate  passed in, as fault
    collapsed, fault collapses it and checks to  see  if  the
    gate's  output  fault  state  has changed.  If the output
    fault state has changed then the change is passed  on  to
    the  gates  that  have  inputs tied to the current gate's
    output.  The function marks these gates as  unfault  col-
    lapsed  because  it changed the fault state of one of the
    inputs.  This function is called by  the  function  'col-
    lapse.'


ARGUMENTS: The function is passed the gate number of the gate
    it is to fault collapse.


RETURN: none.


FUNCTIONS CALLED: none.  */


#include "define.c"

int push( index )

  int index;

{
  extern struct GATES   gates[];
  extern struct NET     net[];
  extern struct LIST    list[];

  int   i;
  long  out;

  list[ index ].stable = 0;
```

```
    /* Call the fault collapsing function associated with the
    gate  in  the  gate library.  If its output has changed a
    nonzero value is returned that represent the  new  fault.
    This  fault  is passed forward towards the primary output
    along the output net of the gate. */

  if ( out = ( *( gates[ list[ index ].offset ].fault ))(
                 &list[ index ].state ))
    for( i = net[ index = list[ index].net ].fanout; i--; )
    {
#     ifdef DEBUG
        printf( "push: assigning faults to gate %d pin %d\n",
                 net[ index ].elements[i], net[index].pin[i]);
#     endif
      list[ net[ index ].elements[ i ] ].stable = 1;
      list[ net[ index ].elements[ i ] ].state |=
           ((long)out << ( 2 * (net[ index ].pin[i]-1)));
    }
}
```

```
/*
                  Queue and Unqueue Functions


                        Joseph W. Naab
                        April 27, 1988
                         Version 1.0


DESCRIPTION:  These are the two functions used by the 'simu-
    late' function to maintain the priority linked list that
    is the event schedule.  The function 'queue' puts gates
    in the queue and 'unqueue' returns the gate that is to be
    simulated next.  The queue is a time ordered linked list
    that contains a bit string that represents each gate.  To
    put a gate in the queue the 'queue' function finds the
    queue element with the same time as the time the gate is
    to be simulated and sets the bit corresponding to the
    gate in the queue element.  If no queue elements exist
    for the time of the gate one is created.  The function
    'unqueue' returns a gate from the top of the queue, the
    gate with the smallest time.  If more than one gate is
    scheduled for simulation at the same time the gate with
    the smallest number is returned.

    To maintain the linked list and their present position in
    the linked list both functions use static variables.  The
    queue is dynamically allocated and to reduce the calls to
    the memory allocation function the unused queue elements
    are kept and reused.  The gates are each represented by a
    bit in a character string.  If the bit corresponding to a
    gate is set in a queue element then that gate is
    scheduled for simulation at the queue element's time.


ARGUMENTS:  The gate number and the time it is to be simulat-
    ed are passed into 'queue'.  The gate number and the time
    it is to be simulated are returned by 'unqueue'.


RETURN: 'Queue' does not return anything.  'unqueue' returns
    a 1 if a gate was found in the queue and a 0 if the queue
    is empty.


FUNCTIONS CALLED:  none.  */


#include "define.c"
```

```
static struct QUEUE *head = NULL, /* top queue linked list */
                    *junk = NULL; /* top of junk pile      */

static int next = 0; /* current character in unqueue          */

int queue( element, time )

  int element,
      time;

{
  extern int   num_gates;

  int i;

  struct QUEUE *temp,
               *old = NULL,
               *current;

# ifdef DEBUG
    printf( "queue: element %d at time %d\n",element, time );
# endif

    /* Look for the queue element with the same time  as  the
    gate time passed in.  */

  for( current = head; current && current->time < time;
              current= current->nexttime)
    old = current;
  if( current && current->time == time )
    current->element[ element / 8 ] |= 1 << ( element % 8 );
  else
  {
    if( junk )
    {
      temp = junk;
      junk = junk->nexttime;
    }
    else
    {

    /* If a queue element does not exist for the gate's  time
    create  one that does.  If it is  an unused queue element
    on the junk pile, use it, if not allocate on form  memory.
    */

      temp = ( struct QUEUE *)malloc( sizeof( struct QUEUE )
                                    + num_gates/8 );
      for( i = 0; i <= num_gates/8; temp->element[i++] = 0 );
    }

    /* Put the gate in the queue element.  Each  bit  in  the
    character  string  corresponds  to a gate.  The gates are
    ordered by the gate number they had in the netlist  file.
    */

    temp->nexttime = current;
    temp->time = time;
    temp->element[ element / 8 ] |= 1 << ( element % 8 );
    if( old )
      old->nexttime = temp;
    else
      head = temp;
  }
}

int unqueue( element, time )

  int *element,
      *time;
{
  struct QUEUE *temp;
  char  c;
  int bit;

    /* While the queue  is  not  empty.  Look  for  the  next
    nonzero character in the queue elements or the end of the
    queue element.  If the end of queue  element  is  reached
    then  put it in the junk pile.  If a nonzero character is
    found find the first gate in the character  and  pass  it
    back to the calling function 'simulate'.  */

  while( head )
  {
    for( ; next <= num_gates/8+1 && !head->element[next];
          next++ );
    if( next > num_gates/8+1 )
    {
      next = 0;
      temp = head;
      head = temp->nexttime;
      temp->nexttime = junk;
      junk = temp;
    }
    else
    {
      c = head->element[next];
      if( c & 15 )
```

```
          if( c & 3 )
            if( c & 1 )
              bit = 0;
            else
              bit = 1;
          else
            if( c & 4 )
              bit = 2;
            else
              bit = 3;
        else
          if( c & 0x30 )
            if( c & 0x10 )
              bit = 4;
            else
              bit = 5;
          else
            if( c & 0x40 )
              bit = 6;
            else
              bit = 7;
        *element = next*8 + bit;
        *time = head->time;
        head->element[ next ] &= -2 << bit;
#       ifdef DEBUG
          printf( "unque: elem %d time %d bit %d next %d\n",
                   *element,*time, bit, next );
#       endif
        return 1;
      }
    }
# ifdef DEBUG
    printf( "unqueue: queue is empty\n " );
# endif
  return 0;
}
```

```
/*
                          Simulate Function


                          Joseph W. Naab
                          April 27, 1988
                           Version 1.0


DESCRIPTION:  This is the primary function that simulates the
    circuit. The function simulates in parallel  n-1  faulted
    circuits and an unfaulted circuit for  the  entire set of
    input  vectors, where n is the number of bits  in  a long
    integer, WORDLENGTH.  At the  completion  of  each  input
    vector the  primary outputs  are examined to see  if  any
    faulted circuits output differs from  the  unfaulted cir-
    cuits output.  If one does and  it  hasn't  already  been
    detected,  a detected fault output statement is generated
    and the faulted circuit is removed from further consider-
    ation by removing its bit from the mask. When the outputs
    are compared the mask  is used to ignore any circuit that
    has already been detected.

    After the simulation of the last  input  vector,  faulted
    circuits  remaining in the mask are not detectable by the
    input vectors.  For each of  these  faulted  circuits  an
    undetected fault message is produced. If the gate the un-
    detected fault occurred on was internal to  a  macro  a
    undetected macro fault output is generated.


ARGUMENTS:  The file pointer to the three  output  files gen-
    erated  by the fault grader are passed input the function
    along with the number of  faulted  circuits  to  actually
    be simulated. Normally n-1 faulted circuits are simulated
    but the last time any number of faulted less than n-1 may
    be all thats left.


RETURN:  The function returns the number of faults that  were
    detected by the input vectors.


FUNCTIONS CALLED:  The  four  functions  called,  'simulate',
    'queue'  and  'unqueue'  are  used  to schedule gates for
    simulation and find the next gate scheduled,  respective-
    ly.  The function 'evaluate' is responsible for the simu-
    lation of the individual gates. 'macrofault' returns  the
    macro  number  and the macro fault message of faults that
```

```
   occur internal to a macro.  */


#include "define.c"

int simulate( detected, undetected, macrofile, faults )

  FILE *detected,
       *undetected,
       *macrofile;
  int  faults;

{
  extern struct FAULT   parallel[];/* info of fauted circs */
  extern struct GATES   gates[];   /* library of gates     */
  extern struct LIST    list[];    /* holds gate's info    */
  extern struct LIST    *data[];   /* list of data inputs  */
  extern struct LIST    *output[]; /* list primary outputs */
  extern struct MACROLIST macrolist[]; /* list of macros   */
  extern struct MACROS  macros[];  /* library of macros    */
  extern struct NET     net[];     /* holds netlist's info */

  extern int        vector[][MAXINPUTS]; /* input vectors  */
  extern int        num_data;      /* # data in model      */
  extern int        num_gates;     /* # gates in model     */
  extern int        num_output;    /* # outputs in model   */
  extern int        num_vectors;   /* # input vectors      */
  extern int        expanded;      /* first gate in macro  */


  int evaluate();
  char *macrofault();
  int queue();
  int unqueue();

  int i,j,index,element,row,time,macro;
  int detectedfaults = 0;
  long logic, defined, mask;
  int *input;
  char *map;

/* Set up the mask to show which bits of the simulation  word
is  associated  with  an active faulted circuit.  If a bit is
not set in mask then the corresponding bit in the  simulation
word is always ignored. */

  mask = ~( -1 << faults );
```

```
/* Give the primary inputs the values of  the  input  vectors
and  queue  the elements that are tied to the primary inputs.
*/

  for( row = 0; row < num_vectors; row++ )
  {
    for( i=0; i <= num_data; i++ )
    {
      index = data[ i ]->net;
      for( j = net[ index ].fanout; j--; )
      {
        element = net[ index ].elements[ j ];
#       ifdef DEBUG
          printf( "simulate: data %3d being queued, net %3d",
                   element, index, " logic %x %3d %3d\n",
                   -vector[i][row], i, row);
#       endif
        queue( element, gates[ list[ element].offset ].time);
        net[ index ].logic = -vector[ i ][ row ];
        net[ index ].defined = -1;
      }
    }

/* Simulate the circuit until the queue is empty, the circuit
has  stabilized.  If the current gate changed its output then
schedule the gates that have inputs tied to the output of the
changing gate. */

    while( unqueue( &element, &time ))
      if( evaluate( element, &time ))
#       ifdef DEBUG
        { printf( "simulate: element %d evaluated", element);
          printf( " and changed to time %d\n", time );
#       endif
        for( j = net[ list[ element ].net ].fanout; j--; )
        {
          index = net[ list[ element ].net ].elements[ j ];

          queue( index, time+gates[list[index].offset].time);

#         ifdef DEBUG
            printf("simulate:element %d ",list[element].net);
            printf( "being queued, net %d\n",index, element);
#         endif
        }
#       ifdef DEBUG
          } else
            printf( "simulate: element %d ",element);
            printf( " evaluated and not changed.\n");
```

```
#       endif

/* After the simulation of each input vector look at the out-
puts for faulted circuits that differ from the unfaulted cir-
cuit. If one is found that has its bit set  in  'mask',  gen-
erate a detected fault output statement and clear the faulted
circuits bit in 'mask'.*/

    for( i = 0; i <= num_output; i++ )
    {
      input = output[ i ]->pins;
      logic = net[ *input ].logic;
      defined = net[ *input ].defined;
#     ifdef DEBUG
        printf( "OUTPUT: %d logic %x defined %x num %d\n",
                 i, logic, defined, i );
#     endif
#undef DEBUG
      if( logic < 0 )
        logic ^= -1;

      logic &= defined;
      logic &= mask;

      for( j=0; logic; j++ )
      {
        if( logic & 1 )
        {
          printf( "%3d %-8s pin %d stuck-at-%d ",
            parallel[j].element,
            gates[ list[ parallel[j].element].offset].name,
            parallel[j].pin, parallel[j].fault );
          printf( "detected by input %3d\n", row + 1 );

          fprintf( detected, "%3d %-8s pin %d stuck-at-%d ",
            parallel[j].element,
            gates[ list[ parallel[j].element].offset].name,
            parallel[j].pin, parallel[j].fault );
          fprintf( detected,"detected by input %3d\n",row+1);

          detectedfaults++;

          mask ^= 1 << j;
#   ifdef DEBUG
      printf( "mask = %x\n", mask );
#   endif
        }
        logic >>= 1;
      }
      }
    }

/* After the simulation of all the input vectors look through
mask  for  faulted circuits that remain undetected.  For each
undetected faulted circuit  generate  an  undetected  faulted
output statement.  If the gate the fault appear on was in the
expanded gates section of the model the gate is internal to a
macro.   So find the macro number and macro fault message and
generate a macro undetected output statement. */

  for( j=0; mask; j++ )
  {
    if( mask & 1 )
    {
        fprintf( undetected, "%3d %-8s pin %d stuck-at-%d ",
            parallel[j].element,
            gates[ list[ parallel[j].element].offset].name,
            parallel[j].pin, parallel[j].fault );
        fprintf( undetected, " remains undetected.\n" );

      if( parallel[ j ].element >= expanded )
      {
        map = macrofault( parallel[j].element,
              parallel[j].pin,  parallel[j].fault, &macro);
        if( map[0] )
        {
          printf( "%3d %-8s %s\n",
            macrolist[macro].number,
            macros[ macrolist[ macro ].offset ].name, map );
          printf( "\t%3d %-8s pin %d stuck-at-%d ",
            parallel[j].element,
            gates[ list[ parallel[j].element].offset].name,
            parallel[j].pin, parallel[j].fault );
          printf( " remains undetected.\n" );

          fprintf( macrofile, "%3d %-8s %s\n",
            macrolist[macro].number,
            macros[ macrolist[ macro ].offset ].name, map );
          fprintf( macrofile,"\t%3d %-8s pin %d stuck-at-%d",
            parallel[j].element,
            gates[ list[ parallel[j].element].offset].name,
            parallel[j].pin, parallel[j].fault );
          fprintf( macrofile, " remains undetected.\n" );
        }
      }
      else
      {
        printf( "%3d %-8s pin %d stuck-at-%d ",
              parallel[j].element,
              gates[ list[ parallel[j].element].offset].name,
              parallel[j].pin, parallel[j].fault );
          printf( " remains undetected.\n" );

          fprintf( macrofile, "%3d %-8s pin %d stuck-at-%d ",
              parallel[j].element,
              gates[ list[ parallel[j].element].offset].name,
              parallel[j].pin, parallel[j].fault );
          fprintf( macrofile, " remains undetected.\n" );
        }
      }
      mask >>= 1;
#     ifdef DEBUG
        printf( "mask = %x\n", mask );
#     endif
    }

    return detectedfaults;
}
```

```
/*
                         Unusedmacros Function

                            Joseph W. Naab
                            April 27, 1988
                              Version 1.0

DESCRIPTION:  This function walks through the macro list 'ma-
    crolist'  and  removes  the parts of compound macros that
    are unused.  A macro part is unused if the the pin speci-
    fied  in  the  macro  library is unconnected, tied to the
    zero net.  By looking at each of these pins the  function
    can  tell  which  parts  of  compound  macros are unused.
    Unused macro parts are removed from the fault grading  by
    removing  any faults that appear on the gates internal to
    the unused part of the macro.


ARGUMENTS: none.


RETURN: none.


FUNCTIONS CALLED: none.  */


#include "define.c"

int unusedmacros()

{
  extern struct LIST       list[];
  extern struct MACROS     macros[];
  extern struct MACROPARTS macroparts[];
  extern struct MACROLIST  macrolist[];

  extern int expanded;
  extern int num_macros;

  int i,
      last,
      first,
      current,
      walker,
      index;

  last = expanded;
```

```
    /* Look at the pin specified in the macro library of each
    part  of compound macros.  If the macro part is unused go
    through the gates used in that macro part  expansion  and
    remove any fault that exist on them. */

  for( current = 0; current < num_macros; current++ )
  {
    walker = last;
    first = last;
    last += macros[ macrolist[ current ].offset ].gates;

    if( index = macros[ macrolist[current].offset ].parts )
      while( walker < last )
      {
        if( !*( list[ first + macroparts[index].gate ].pins
                + macroparts[ index ].pin - 1 ))
          for( i = macroparts[ index ].length; i; i-- )
            list[ walker + i ].state = 0;

        walker += macroparts[ index++ ].length;
      }
   }
}
```

```
/*
                         Vector_input Function

                            Joseph W. Naab
                            April 27, 1988
                             Version 1.0

DESCRIPTION:  This function reads the file 'filename.VEC' and
    fills  the  array vector.  The file contains only 1's and
    0's.  Each row corresponds to an input  vector  and  each
    column  corresponds  to  a primary input, DATA component.
    The first row is the first input  vector  and  the  first
    column is DATA1.

ARGUMENTS: A file  pointer  to  the  file  'filename.VEC'  is
    passed to the function.

RETURN: The number of input vectors read is returned  to  the
    calling function 'main'.

FUNCTIONS CALLED: none.  */

#include "define.c"

int input_vector( file )

  FILE *file;

{
  extern int   vector[][ MAXINPUTS ];
  extern int   num_data;

  char inbuf[ 80 ];
  int row =0, i;

  row = 0;
  while( fgets( inbuf, 80, file ))
  {
    for( i = 0; i <= num_data; i++ )
      vector[ i ][ row ] = inbuf[ i ] - '0';
    row++;

  }
  return row;
}
```

Appendix C

Gates Library

by

Joseph W. Naab

April 27, 1988

```
/*
                            Gates Library


                           Joseph W. Naab
                           April 27, 1988
                             Version 1.0


DESCRIPTION: This is the gate library and consists of a
  structure entry for each gate and the functions that per-
  form fault collapsing and simulation of each gate. The
  structure initialization appears at the bottom of the
  file and consists of the gate's name, the number of pins,
  the time delay of the gate, the name of the function used
  for fault compression, and the name of the function used
  for simulation. The name of the gate can not exceed
  eight characters because of the input specifications of
  the fault grader. A gate can have any number of pins up
  to sixteen, but only one of them may be an output. The
  time delay of the gate must be an integer and is used by
  the simulator to determine at what time the gate's output
  will change because of an input value change.

  The fault collapsing function is responsible for applying
  the fault dominance and equivalence rules for the gate.
  The AND gate's fault collapsing function will be the only
  fault collapsing function commented. All the fault func-
  tions pass and return the same types of information. The
  input argument 'state' represents the faults by using
  each bit of the word 'state' to represent one of the pos-
  sible faults. The least significant two bits represent
  the faults on the first input, the next two bits
  represent the next input and so on until the output's
  faults are represented. The least significant bit in
  each pair represents the stuck-at-zero fault. The other
  bit represents the stuck-at-one fault. A fault exists if
  the bit representing it is set.

  The values returned signify the output fault state. If a
  zero is returned the output state remains unchanged, if a
  1 is returned the output state of the gate has changed to
  a stuck-at-zero fault. Returning a value of two would in-
  dicate that the output had acquired a stuck-at-one fault,
  and a returned value of three would represent both
  faults. It is necessary to return the new fault value of
  the output so it can be propagated forward through the
  circuit.
```

```
   The arguments input to all the simulation  functions  are
   also  the same. The first two are pointers to integer ar-
   rays that hold the values of the inputs.   The  subscript
   zero  refers  to  the first pin, the subscript one to the
   second input, and so on for each of the pins.  The  array
   pointed to by the pointer 'logic_input' contains the log-
   ical values of  the  inputs.  The  array  pointed  to  by
   'define_input'  contains  the defined state of the input.
   A zero in the 'define_input' array means that the logical
   value  for  that  input  is not defined.  No value is re-
   turned by the simulation functions.

   The procedure for adding another gate to the library fol-
   lows:

        - Write a fault function for the gate.
        - Write a simulation function for the gate.
        - Add the gate to the initialization list of the
          structure 'GATES'.
        - Increase the value 'GATE_NUMBER' by one for each
          gate added to the library.
        - Recompile and relink this file.
*/

NOTE:  This is only a partial listing, see the file
       'gates.c' for the entire library.

#include "define.c"

/* AND2f AND2f AND2f AND2f AND2f AND2f AND2f AND2f AND2f */
int and2f( state )

  long *state;
{
  /* If the output already has a stuck-at-0 fault then  re-
  move  any  stuck-at-0 faults on the inputs and return the
 status output unchanged. */

  if( !( *state & 0x10 ))

  /* Check to see if one of the  inputs  has  a  stuck-at-0
  fault.*/

    if( *state & 0x5 )
    {

  /* There is a stuck-at-0 on one of the inputs so generate
  a  stuck-at-0 on the output and return a output status of
  changed to stuck-at-0 */
```

```
        *state |= 0x10;
        *state &= 0x3a;
        return 1;
      }
    *state &= 0x3a;
    return 0;
}

/*AND2S AND2S AND2S AND2S AND2S AND2S AND2S AND2S AND2S  */
int and2s( logicin, definein, logicout, defineout )

    int *logicin,
        *definein,
        *logicout,
        *defineout;

{
    /* Determine if the output of the AND gate is defined  or
    not.   Translates to either input with a defined zero in-
    put or both inputs defined causes the output  to  be  de-
    fined. */

    *defineout = ( definein[0] & ~logicin[0] )
                   | ( definein[1] & ~logicin[1] )
                   | ( definein[0] & definein[1] );

    /* Logical AND of the two inputs regardless of thier  de-
    fined states. */

    *logicout = ( logicin[0] & logicin[1] );
}

/* AND9F AND9F AND9F AND9F AND9F AND9F AND9F AND9F AND9F */
int and9f( state )

    long *state;

{

    if( !( *state & 0x40000 ))
      if( *state & 0x15555 )
      {
        *state |= 0x40000;
        *state &= 0xeaaaa;
        return 1;
      }
    *state &= 0xeaaaa;
    return 0;
}
```

```
/* AND9S  AND9S  AND9S  AND9S  AND9S  AND9S  AND9S  AND9S*/
int and9s( logicin, definein, logicout, defineout )

  long *logicin,
       *definein,
       *logicout,
       *defineout;

{
  *defineout = ( definein[0] & ~logicin[0] )
                   | ( definein[1] & ~logicin[1] )
                   | ( definein[2] & ~logicin[2] )
                   | ( definein[3] & ~logicin[3] )
                   | ( definein[4] & ~logicin[4] )
                   | ( definein[5] & ~logicin[5] )
                   | ( definein[6] & ~logicin[6] )
                   | ( definein[7] & ~logicin[7] )
                   | ( definein[8] & ~logicin[8] )
                   | ( definein[0] & definein[1] &
                       definein[2] & definein[3] &
                       definein[4] & definein[5] &
                       definein[6] & definein[8] &
                       definein[9] );

  *logicout = ( logicin[0] & logicin[1] & logicin[2] &
                logicin[3] & logicin[4] & logicin[5] &
                logicin[6] & logicin[8] & logicin[9] );
}

/* OR2F OR2F OR2F OR2F OR2F OR2F OR2F OR2F OR2F OR2F OR2F*/
int or2f( state )
  long *state;
{
  if( !( *state & 0x20 ))
    if( *state & 0xa )
    {
      *state |= 0x20;
      *state &= 0x35;
      return 2;
    }
  *state &= 0x35;
  return 0;
}

/* OR2S  OR2S  OR2S  OR2S  OR2S  OR2S  OR2S  OR2S  OR2S */
int or2s( logicin, definein, logicout, defineout )

  long *logicin,
       *definein,
       *logicout,
       *defineout;

{
  *defineout = ( definein[0] & logicin[0] )
                 | ( definein[1] & logicin[1] )
                 | ( definein[0] & definein[1] );

  *logicout = ( logicin[0] | logicin[1] );
}

/* INVF INVF INVF INVF INVF INVF INVF INVF INVF INVF INVF*/
int invf( state )
  long *state;
{
  if( !( *state & 0xc ))
    if( *state & 0x3 )
    {
      *state |= *state << 2;
      *state &= 0xc;
      return *state >> 2;
    }
  *state &= 0xc;
  return 0;
}

/* INVS  INVS  INVS  INVS  INVS  INVS  INVS  INVS  INVS  */
int invs( logicin, definein, logicout, defineout )

  long *logicin,
       *definein,
       *logicout,
       *defineout;

{
  *defineout = definein[0];

  *logicout = ~logicin[0];
}

/* BUFFERF BUFFERF BUFFERF BUFFERF BUFFERF BUFFERF */
int bufferf( state )
  long *state;
{
  if( !( *state & 0xc ))
    if( *state & 0x3 )
    {
      *state |= *state << 2;
      *state &= 0xc;
      return *state >> 2;
    }
  *state &= 0xc;
  return 0;
}

/* EXOR2F EXOR2F EXOR2F EXOR2F EXOR2F EXOR2F EXOR2F */
int exor2f( state )

  long *state;

{
# ifdef DEBUG
    printf( "exor2f: in state %x ", *state );
    printf( "exor2f: out state %x0, *state );
# endif
  return 0;
}

/* EXOR2S EXOR2S EXOR2S EXOR2S EXOR2S EXOR2S EXOR2S */
int exor2s( logicin, definein, logicout, defineout )

  long *logicin,
       *definein,
       *logicout,
       *defineout;

{
  *defineout =  ( definein[0] & definein[1] );

  *logicout = ( logicin[0] ^ logicin[1] );
}


#define GATE_NUMBER  41  /* Number of members in GATES */

struct GATES gates[ GATE_NUMBER + 1 ] =
{
 /* gate        pin gate      fault   simulate
    name        #   delay  function   function */

    "INV",      2,  1,        invf,      invs,
    "NAND2",    3,  1,      nand2f,    nand2s,
    "AND2",     3,  1,       and2f,     and2s,
    "OR2",      3,  1,        or2f,      or2s,
    "NOR2",     3,  1,       nor2f,     nor2s,
    "EXOR2",    3,  1,      exor2f,    exor2s,
    "NAND3",    4,  1,      nand3f,    nand3s,
    "AND3",     4,  1,       and3f,     and3s,
        "OR3",      4,  1,        or3f,     or3s,
        "NOR3",     4,  1,       nor3f,    nor3s,
        "EXOR3",    4,  1,      exor3f,   exor3s,
        "BUFFER",   2,  1,     bufferf,  buffers,
        "DATA1",    1,  1,       dataf,    datas,
        "DATA2",    1,  1,       dataf,    datas,
        "DATA3",    1,  1,       dataf,    datas,
        "DATA4",    1,  1,       dataf,    datas,
        "DATA5",    1,  1,       dataf,    datas,
        "DATA6",    1,  1,       dataf,    datas,
        "DATA7",    1,  1,       dataf,    datas,
        "DATA8",    1,  1,       dataf,    datas,
        "DATA9",    1,  1,       dataf,    datas,
        "DATA10",   1,  1,       dataf,    datas,
        "DATA11",   1,  1,       dataf,    datas,
        "DATA12",   1,  1,       dataf,    datas,
        "DATA13",   1,  1,       dataf,    datas,
        "DATA14",   1,  1,       dataf,    datas,
        "DATA15",   1,  1,       dataf,    datas,
        "DATA16",   1,  1,       dataf,    datas,
        "OUTPUT",   1,  1,       dataf,    datas,
        "NAND4",    5,  1,      nand4f,   nand4s,
        "AND4",     5,  1,       and4f,    and4s,
        "OR4",      5,  1,        or4f,     or4s,
        "NOR4",     5,  1,       nor4f,    nor4s,
        "NAND5",    6,  1,      nand5f,   nand5s,
        "AND5",     6,  1,       and5f,    and5s,
        "OR5",      6,  1,        or5f,     or5s,
        "NOR5",     6,  1,       nor5f,    nor5s,
        "NAND9",   10,  1,      nand9f,   nand9s,
        "AND9",    10,  1,       and9f,    and9s,
        "OR9",     10,  1,        or9f,     or9s,
        "NOR9",    10,  1,       nor9f,    nor9s
};

    /* Construct an extern variable number of gates*/

int gate_library = GATE_NUMBER;
```

Appendix D

Macros Library

by

Joseph W. Naab

April 27, 1988

```
/*
                              Macros Library


                              Joseph W. Naab
                              April 27, 1988
                               Version 1.0


DESCRIPTION:  This is a library that defines the macros and
  provides the messages that map the faults internal to the
  macros to the periphery of the macro.   Another  function
  of this library is to provide information that is used to
  remove unused parts of compound  macros  form  the  fault
  grading.   A  description of the methods for adding a new
  macro are explained in the thesis "A Digital Logic  Fault
  Grader" written by Joe Naab for the Electrical and Compu-
  ter Engineering Department  at  Kansas  State  University.


  The library consists  of  three  structures.   The  first
  structure  contains  the  fault mapping message for every
  possible internal fault associated with the macro.  These
  messages  are  intended  to  give  a  relationship of the
  internal fault to a external fault or  condition  of  the
  macro.  It is very important that every possible fault be
  included in the macros definition.  If  one  is  excluded
  the  program  will be forced to abort when the fault mes-
  sage is not found.  Some of the  faults  associated  with
  the macro are redundant faults and can not be detected by
  nondynamic testing.  These faults can be removed from the
  fault grading by giving it a null fault message, ie. "".


  The second macro 'macros' gives the name  of  the  macro,
  less  than  eight characters, the number of gates used in
  the macro's expansion, the number of internal faults pos-
  sible,  a offset into the fault message structure, and an
  offset into the macro parts structure.  The third  struc-
  ture, 'macroparts'  identifies the the parts of compound
  macros.  An example is the 7474 which has two edge  trig-
  gered  D  flip-flops.   The  structure contains three in-
  tegers, the first two identify the gate and pin that will
  have a 0 net number if the part is unused.  The third in-
  teger is the number of gates that make up  this  part  of
  the  macro.   Each  part  of a compound macro is examined
  after the construction  of  the  collapsed  faults.   Any
  unused  macro  parts are removed from the fault grader by
  deleting any faults that  appear  on  the  gates  of  the
  unused macro part. */

#include "define.c"


struct MACROFAULTS macrofaults[] =
{
/* 7474 Edge triggered D-flipflop  */

/* 1.1 */ 1, 1, 1, "Preset stuck-at-one fault",
/* 1.2 */ 1, 2, 1, "D input stuck-at-zero fault.",
/* 1.3 */ 1, 3, 1, "Clear stuck-at-zero fault.",
/* 1.4 */ 2, 1, 1, "Flipflop ability hold clear thru ck",
/* 1.5 */ 2, 2, 1, "Clear stuck-at-one fault Cl while Ck ",
/* 1.6 */ 2, 3, 1, "Clock stuck-at-one fault.",
/* 1.7 */ 3, 1, 1, "Clock or Clear stuck-at-zero fault",
/* 1.8 */ 3, 2, 1, "Clock stuck-at-one fault.",
/* 1.9 */ 3, 3, 1, "",
/* 1.10*/ 4, 1, 1, "Ability to hold clear ck high.",
/* 1.11*/ 4, 2, 1, "",
/* 1.12*/ 4, 3, 1, "D stuck-at-one fault.",
/* 1.13*/ 5, 1, 1, "Preset control of primary output",
/* 1.14*/ 5, 2, 1, "Clear or Clock stuck-at-zero fault.",
/* 1.15*/ 5, 3, 1, "Flip-flop ability to hold set",
/* 1.16*/ 6, 1, 1, "Preset stuck-at-zero fault.",
/* 1.17*/ 6, 2, 1, "Clear control over secondary output",
/* 1.18*/ 6, 3, 1, "Flip-flop ability to hold clear",
/* 1.19*/ 7, 1, 1, "Preset stuck-at-one fault.",
/* 1.20*/ 7, 2, 1, "D input stuck-at-zero fault.",
/* 1.21*/ 7, 3, 1, "Clear stuck-at-zero fault.",
/* 1.22*/ 8, 1, 1, "Flipflop ability hold clear thru ck",
/* 1.23*/ 8, 2, 1, "Clear stuck-at-one fault Cl while Ck ",
/* 1.24*/ 8, 3, 1, "Clock stuck-at-one fault.",
/* 1.25*/ 9, 1, 1, "Clock or Clear stuck-at-zero fault",
/* 1.26*/ 9, 2, 1, "Clock stuck-at-one fault.",
/* 1.27*/ 9, 3, 1, "",
/* 1.28*/ 10, 1, 1, "Ability to hold clear ck high.",
/* 1.29*/ 10, 2, 1, "",
/* 1.30*/ 10, 3, 1, "D stuck-at-one fault.",
/* 1.31*/ 11, 1, 1, "Preset control of primary output",
/* 1.32*/ 11, 2, 1, "Clear of Clock stuck-at-zero fault.",
/* 1.33*/ 11, 3, 1, "Flip-flop ability to hold set",
/* 1.34*/ 12, 1, 1, "Preset stuck-at-zero fault.",
/* 1.35*/ 12, 2, 1, "Clear control of secondary output",
/* 1.36*/ 12, 3, 1, "Flip-flop ability to hold clear",

/*  74138, 3 X 8 Decoder/Demultiplexor */

/* 2.1 */  1, 1, 1, "Select A stuck-at-0 for output 0.",
/* 2.2 */  1, 2, 1, "Select B stuck-at-0 for output 0.",
/* 2.3 */  1, 3, 1, "Select C stuck-at-0 for output 0.",
/* 2.4 */  1, 4, 1, "Ability to disable output 0.",
/* 2.5 */  2, 1, 1, "Select A stuck-at-1 for output 1.",
/* 2.6 */  2, 2, 1, "Select B stuck-at-0 for output 1.",
/* 2.7 */  2, 3, 1, "Select C stuck-at-0 for output 1.",
/* 2.8 */  2, 4, 1, "Ability to disable output 1.",
/* 2.9 */  3, 1, 1, "Select A stuck-at-0 for output 2.",
/* 2.10*/  3, 2, 1, "Select B stuck-at-1 for output 2.",
/* 2.11*/  3, 3, 1, "Select C stuck-at-0 for output 2.",
/* 2.12*/  3, 4, 1, "Ability to disable output 2.",
/* 2.13*/  4, 1, 1, "Select A stuck-at-1 for output 3.",
/* 2.14*/  4, 2, 1, "Select B stuck-at-1 for output 3.",
/* 2.15*/  4, 3, 1, "Select C stuck-at-0 for output 3.",
/* 2.16*/  4, 4, 1, "Ability to disable output 3.",
/* 2.17*/  5, 1, 1, "Select A stuck-at-0 for output 4.",
/* 2.18*/  5, 2, 1, "Select B stuck-at-0 for output 4.",
/* 2.19*/  5, 3, 1, "Select C stuck-at-1 for output 4.",
/* 2.20*/  5, 4, 1, "Ability to disable output 4.",
/* 2.21*/  6, 1, 1, "Select B stuck-at-0 for output 5.",
/* 2.22*/  6, 2, 1, "Select C stuck-at-1 for output 5.",
/* 2.23*/  6, 3, 1, "Select A stuck-at-1 for output 5.",
/* 2.24*/  6, 4, 1, "Ability to disable output 5.",
/* 2.25*/  7, 1, 1, "Select A stuck-at-0 for output 6.",
/* 2.26*/  7, 2, 1, "Select C stuck-at-1 for output 6.",
/* 2.27*/  7, 3, 1, "Select B stuck-at-1 for output 6.",
/* 2.28*/  7, 4, 1, "Ability to disable output 6.",
/* 2.29*/  8, 1, 1, "Select B stuck-at-1 for output 7.",
/* 2.30*/  8, 2, 1, "Select A stuck-at-1 for output 7.",
/* 2.31*/  8, 3, 1, "Select C stuck-at-1 for output 7.",
/* 2.32*/  8, 4, 1, "Ability to disable output 7.",
/* 2.33*/  9, 1, 0, "G2A stuck-at-0.",
/* 2.34*/  9, 2, 0, "G2B stuck-at-0.",
/* 2.35*/  9, 3, 0, "G1 stuck-at-1.",

/* 74194 4-bit Shift Register */

/* 3.1 */  1, 1, 1, "Data input A stuck-at-1.",
/* 3.2 */  1, 2, 1, "Can't ignore Data A's 1.",
/* 3.3 */  2, 1, 1, "Shift Right Input stuck-at-1.",
/* 3.4 */  2, 2, 1, "Can't ignore Shift Right Input 1.",
/* 3.5 */  3, 1, 0, "Can't right shift a 1 into QA.",
/* 3.6 */  3, 2, 0, "Can't load a 1 into QA.",
/* 3.7 */  3, 3, 0, "Can't left shift a 1 into QA.",
/* 3.8 */  5, 1, 1, "Can't left shift 0 into QA.",
/* 3.9 */  5, 2, 1, "QA can't ignore QB's 1.",
/* 3.10*/  6, 1, 1, "Can't right shift 0 into QB.",
/* 3.11*/  6, 2, 1, "Can't ignore Data B's 1.",
/* 3.12*/  7, 1, 1, "Can't left shift 0 into QB.",
/* 3.13*/  7, 2, 1, "Can't ignore QA's 1.",
/* 3.14*/  8, 1, 0, "Can't right shift a 1 into QB.",
/* 3.15*/  8, 2, 0, "Can't load a 1 into QB.",
/* 3.16*/  8, 3, 0, "Can't left shift a 1 into QB.",
/* 3.17*/  9, 1, 1, "Can't left shift 0 into QB.",
/* 3.18*/  9, 2, 1, "QB can't ignore QC's 1.",
/* 3.19*/ 10, 1, 1, "Can't right shift 0 into QC.",
/* 3.20*/ 10, 2, 1, "Can't ignore Data C's 1.",
/* 3.21*/ 11, 1, 1, "Can't left shift 0 into QC.",
/* 3.22*/ 11, 2, 1, "Can't ignore QB's 1.",
/* 3.23*/ 12, 1, 0, "Can't right shift a 1 into QC.",
/* 3.24*/ 12, 2, 0, "Can't load a 1 into QC.",
/* 3.25*/ 12, 3, 0, "Can't left shift a 1 into QC.",
/* 3.26*/ 13, 1, 1, "Can't left shift 0 into QC.",
/* 3.27*/ 13, 2, 1, "QC can't ignore QD's 1.",
/* 3.28*/ 14, 1, 1, "Can't right shift 0 into QC.",
/* 3.29*/ 14, 2, 1, "Can't ignore Data D's 1.",
/* 3.30*/ 15, 1, 1, "Can't left shift 0 into QD.",
/* 3.31*/ 15, 2, 1, "Can't ignore QC's 1.",
/* 3.32*/ 16, 1, 0, "Can't right shift a 1 into QD.",
/* 3.33*/ 16, 2, 0, "Can't load a 1 into QD.",
/* 3.34*/ 16, 3, 0, "Can't left shift a 1 into QD.",
/* 3.35*/ 17, 1, 1, "Can't left shift 0 into QD.",
/* 3.36*/ 17, 2, 1, "QD can't ignore Shift Left Input's 1",
/* 3.37*/ 18, 1, 0, "Ability to load input, S1",
/* 3.38*/ 18, 2, 0, "Ability to load input, S1",
/* 3.39*/ 20, 1, 0, "Ability to hold output, S1",
/* 3.40*/ 20, 2, 0, "Ability to hold output, S0",
/* 3.41*/ 21, 1, 1, "Ability to clock chip",
/* 3.42*/ 21, 2, 1, "Ability to block clock",
/* 3.43*/ 29, 1, 1, "QA indeterminent on set.",
/* 3.44*/ 29, 2, 1, "QA ability to ignore reset on low ck",
/* 3.45*/ 30, 1, 1, "QA ability to ignore set on high ck.",
/* 3.46*/ 30, 2, 1, "QA always reseting.",
/* 3.47*/ 31, 1, 0, "QA ability to hold set pulse high ck",
/* 3.48*/ 31, 2, 0, "QA ability to reset.",
/* 3.49*/ 32, 1, 1, "QA always setting.",
/* 3.50*/ 32, 2, 1, "QA ability to ignore reset high ck.",
/* 3.51*/ 33, 1, 1, "QA ability to ignore set low ck.",
/* 3.52*/ 33, 2, 1, "QA indeterminent on reset.",
/* 3.53*/ 33, 3, 1, "QA ability to stay cleared.",
/* 3.54*/ 34, 1, 0, "QA ability to set.",
/* 3.55*/ 34, 2, 0, "QA ability to hold reset pulse.",
/* 3.56*/ 35, 1, 1, "",
/* 3.57*/ 35, 2, 1, "QA indeterminent on reset.",
/* 3.58*/ 36, 1, 1, "QA ability to set.",
/* 3.59*/ 36, 2, 1, "QA ability to hold reset.",
/* 3.60*/ 37, 1, 1, "QA ability to hold set.",
/* 3.61*/ 37, 2, 1, "QA ability to reset.",
/* 3.62*/ 37, 3, 1, "QA ability to clear.",
/* 3.63*/ 38, 1, 1, "QB indeterminent on set.",
/* 3.64*/ 38, 2, 1, "QB ability to ignore reset on low ck",
/* 3.65*/ 39, 1, 1, "QB ability to ignore set on high ck.",
/* 3.66*/ 39, 2, 1, "QB always reseting.",
/* 3.67*/ 40, 1, 0, "QB ability to hold set pulse high ck",
/* 3.68*/ 40, 2, 0, "QB ability to reset.",
/* 3.69*/ 41, 1, 1, "QB always setting.",
/* 3.70*/ 41, 2, 1, "QB ability to ignore reset high ck.",
/* 3.71*/ 42, 1, 1, "QB ability to ignore set low ck.",
/* 3.72*/ 42, 2, 1, "QB indeterminent on reset.",
/* 3.73*/ 42, 3, 1, "QB ability to stay cleared.",
/* 3.74*/ 43, 1, 0, "QB ability to set.",
/* 3.75*/ 43, 2, 0, "QB ability to hold reset pulse.",
/* 3.76*/ 44, 1, 1, "",
/* 3.77*/ 44, 2, 1, "QB indeterminent on reset.",
/* 3.78*/ 45, 1, 1, "QB ability to set.",
/* 3.79*/ 45, 2, 1, "QB ability to hold reset.",
/* 3.80*/ 46, 1, 1, "QB ability to hold set.",
/* 3.81*/ 46, 2, 1, "QB ability to reset.",
/* 3.82*/ 46, 3, 1, "QB ability to clear.",
/* 3.83*/ 27, 1, 1, "QC indeterminent on set.",
/* 3.84*/ 27, 2, 1, "QC ability to ignore reset on low ck",
/* 3.85*/ 38, 1, 1, "QC ability to ignore set on high ck.",
/* 3.86*/ 38, 2, 1, "QC always reseting.",
/* 3.87*/ 39, 1, 0, "QC ability to hold set pulse high ck",
/* 3.88*/ 39, 2, 0, "QC ability to reset.",
/* 3.89*/ 30, 1, 1, "QC always setting.",
/* 3.90*/ 30, 2, 1, "QC ability to ignore reset high ck.",
/* 3.91*/ 31, 1, 1, "QC ability to ignore set low ck.",
/* 3.92*/ 31, 2, 1, "QC indeterminent on reset.",
/* 3.93*/ 31, 3, 1, "QC ability to stay cleared.",
/* 3.94*/ 32, 1, 0, "QC ability to set.",
/* 3.95*/ 32, 2, 0, "QC ability to hold reset pulse.",
/* 3.96*/ 33, 1, 1, "",
/* 3.97*/ 33, 2, 1, "QC indeterminent on reset.",
/* 3.98*/ 34, 1, 1, "QC ability to set.",
/* 3.99*/ 34, 2, 1, "QC ability to hold reset.",
/* 3.00*/ 35, 1, 1, "QC ability to hold set.",
/* 3.01*/ 35, 2, 1, "QC ability to reset.",
/* 3.02*/ 35, 3, 1, "QC ability to clear.",
/* 3.03*/ 26, 1, 1, "QD indeterminent on set.",
/* 3.04*/ 26, 2, 1, "QD ability to ignore reset on low ck",
/* 3.05*/ 37, 1, 1, "QD ability to ignore set on high ck.",
/* 3.06*/ 37, 2, 1, "QD always reseting.",
/* 3.07*/ 38, 1, 0, "QD ability to hold set pulse high ck",
/* 3.08*/ 38, 2, 0, "QD ability to reset.",
/* 3.09*/ 39, 1, 1, "QD always setting.",
/* 3.10*/ 39, 2, 1, "QD ability to ignore reset high ck.",
/* 3.11*/ 30, 1, 1, "QD ability to ignore set low ck.",
/* 3.12*/ 30, 2, 1, "QD indeterminent on reset.",
/* 3.13*/ 30, 3, 1, "QD ability to stay cleared.",
/* 3.14*/ 31, 1, 0, "QD ability to set.",
/* 3.15*/ 31, 2, 0, "QD ability to hold reset pulse.",
/* 3.16*/ 32, 1, 1, "",
/* 3.17*/ 32, 2, 1, "QD indeterminent on reset.",
/* 3.18*/ 33, 1, 1, "QD ability to set.",
/* 3.19*/ 33, 2, 1, "QD ability to hold reset.",
/* 3.20*/ 34, 1, 1, "QD ability to hold set.",
/* 3.21*/ 34, 2, 1, "QD ability to reset.",
/* 3.22*/ 34, 3, 1, "QD ability to clear.",
};

#define MACRO_NUMBER 2
int macro_library = MACRO_NUMBER;

struct MACROS macros[ MACRO_NUMBER ] =
{
/* macro's  number   number   offset          offset
   name     of       of       into            into
            gates    faults   macrofaults     macroparts */
   "74",    12,      36,      0,              1,
  "138",    16,      35,      35,             0,
  "194",    68,     122,      70,             0,

};


struct MACROPARTS macroparts[ 3 ] =
{
/* gate    pin      number
   number  number   of gates */
   0,      0,       0,
   1,      1,       6,
   7,      1,       6

};
```

Appendix E

References

by

Joseph W. Naab

May 15, 1988

"References"

1. Micro-Logic II, Spectrum Software, 1987

2. Frank F. Tsui, LSI/VLSI Testability Design, McGraw-Hill Book Company, New York, 1987.

3. Statements by D.H. Lenhert, Professor of Electrical and Computer Engineering, Kansas State University, Manhattan, Kansas, April 1988.

4. Alexander Miczo, Digital Logic Testing and Simulation, Harper & Row, Publishers, New York, 1986.

5. Melvin A. Breuer and Arthur D. Friedman, Diagnosis & Reliable Design of Digital Systems, Computer Science Press, Inc., Rockville, Maryland, 1976.

A Digital Logic Fault Grader

by

JOSEPH W. NAAB

BSEE, Kansas State University, 1986

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhatten, KANSAS

1988

ABSTRACT

This thesis describes a software program that evaluates the ability of a set of input vectors to detect digital logic circuit failures, a process that is commonly called fault grading. This fault grading program uses a gate level model of the circuit, the stuck-at-x fault model and an event driven parallel fault simulator to evaluate the input test vectors. Because it was designed to be used by one-time student users working with board sized circuits an effort was made to make the fault grader easy to use and understand. The fault grader's inputs are compatiable with the outputs of a graphic driven digitial logic design package called Micro Logic 2. Both programs operate at the gate level and are easy to understand and use. A documented source code listing and a user's manual are also included in the appendices.