LTEST 1.0

August 1991

Abstract

This is the documentation for the **LTEST 1.0** software package, written for a *Digital Testing and Testable Design* course.

Contents

1 Introduction

The **LTEST** system has been designed as a tool to help students in the *Digital Testing and Testable Design* course. It's objectives are to introduce the concepts of simulation and test generation with easily readable and understandable software. The student can achieve a better understanding of the course material with this system as it combines theoretical explanations along with software used in the field of testing.

The system is a combination of three major building blocks, the circuit parser, the test generator and the simulator. The circuit parser is responsible for reading the circuit description and initiating most of the global data structures. The test generator handles all fault collapsing and test generation assignments. The simulator is responsible for both good circuit simulation and fault simulation. All three blocks are further broken down into submodules to achieve a highly modular design in which every module can be taken out and replaced without disrupting the whole system.

Some basic design policies and the documentation format is presented here to help the student understand the system better.

Procedures and functions communicate by global variables. There are three types of global variable files, "h", "g" and "e". The "h" file contains all constant and type definitions, the "g" file contains the structure declarations and the "e" file contains external declarations for the structures declared in the "g" file. The global variables shared by all modules are defined and declared in the *global* group (*global.h, global.g, global.e*). Other groups include the *d_alg* group (D algorithm variables), *fc* group (fault collapsing variables) and *sim* group (simulation variables).

When describing the global variables, besides their purpose and structure, all procedures which read the variable and all procedures which assign values to the variable are listed. When describing function and procedures, all variables read by and all variables written to by the procedure are listed. Also all functions called by and all functions calling the procedure are documented in the appendices. Thus, a complete cross reference of function-variable and function-function interaction is provided.

Every function is essentially an implementation of an algorithm. The function description is an explanation of its purpose and the step by step description of the algorithm it implements. Every step is assigned a number. The algorithm also appears in the software and the code follows the algorithm to the letter. The numbers representing the algorithm steps appear before each line or group of lines implementing that particular step.

This document proceeds as follows. In Section 2 all global variables are described. Section 3 describes the circuit parser. Section 4 describes the test generation routines, Section 5 describes the good circuit simulator ans Section 6 describes the fault simulator. Section 7 is the user manual. Appendix A is the circuit input grammar. Appendix B is the *function-function* and *function-variable* reference. For each function, the file it is in, all the functions it calls, all the function that call it, all the variables it reads and all the variables it assigns values to are listed. Appendix **??** is the *variable-function* reference. For each global variable the file it is declared in, all functions reading the variable and all functions assigning values to the variable are listed.

2 Global variables

All variables and structures that are used by the three blocks (the *global* group) are first described. Then the variables and structures used by the D algorithm routines (*d_alg* group), the fault collapsing routines (*fc* group) and the good circuit and fault simulators (*sim* group) are described.

For every group, all constants and data types, as found in the "h" file are described, and then the description of the actual variables is provided.

2.1 System global variables

2.1.1 Constants

boolean constants :

false 0

true 1

fail 0

success 1

string and pointer constants :

eos 0

null 0

circuit size parameters :

logic_values 5

max_gate_number 100

max_fan 6

max_line_number max_gate_number*max_fan / 2

max_clocks 6

max_clock_gates max_gate_number / 4

x_ref_length 2*max_line_number - 1

max_timesteps 1000

In order to limit list manipulation and dynamic storage allocation, arrays are used. Since these are allocated before the circuit is read, the circuit size parameters define the maximum allowed values for these parameters. In case some parameter exceeds its maximum allowed value, a message is generated and the maximum value must be changed and the whole system recompiled.

The circuit uses 5 logic values - *0, 1, x, d, dbar*. The maximum number of gates allowed is 100 and the maximum fan-in or fan-out per gate is 6. Since every line is an output of one gate and an input of another, a bound on the number of lines in the circuit is $max\_gate\_number * max\_fan/2$. The maximum number of clocks in the circuit is 6, and a clock can drive no more than $max\_gate\_number/4$ elements. *x_ref_length* is the length of the *Hash* table for line names and *max_timesteps* is the maximum number of value changes per line per simulation.

fault type constants :

sa_0 0

sa_1 1

fault_free 2

logic value constants :

x 3

d 4

dbar 5

The circuit uses 5 logic values. 0 and 1 represent logic values 0 and 1. 3, 4 and 5 represent the values *x, d* and *dbar.*

gate type constants :

and 0

or 1

inv 2

nand 3

nor 4

xor 5

buf 6

dff 7

jkff 8

pi 9

po 10

The system recognizes 11 gate types. *inv* is an inverter, *buf* is a single input multiple output buffer, and *pi* and *po* are primary input and primary output.

delay value constants :

type_rise 0

type_fall 0

type_hold 0

type_setup 0

All 11 gate types have a rise and fall delay (0-1 and 1-0 transition delay). Clocked elements also have hold and setup requirements. All parameters have default values of 0.

2.1.2 Data types

The circuit is modeled as a directed graph. The gates, primary inputs and primary outputs are the graph nodes. The lines are the graph edges. Therefore, two data types are required to represent nodes and edges.

Nodes are represented as

```
typedef struct node
   { char user_name[8] ;
     int q ;
     int qnot ;
     int logic_count[logic_values] ;
     int rise_delay ;
     int fall_delay ;
     int setup_time ;
     int hold_time ;
     int marked ;
     int level ;
   } gate ;
```

The fields of the structure are :

user_name A 7 character user name for the gate.

q all non-clocked gates have either a single input or a single output. The number of that line is stored in this field. For a clocked element, the number of the line connected to the *q* output is stored in the field.

qnot Used only for clocked elements. Holds the number of the line connected to the *qnot* output.

logic_count An array of size *logic_values*. For each value it stores the number of inputs having that value. Used for calculating the output value of a non-clocked element.

rise_delay The 0-1 transition delay of the element.

fall_delay The 1-0 transition delay of the element.

setup_time The required setup time for the element. Used only for clocked elements.

hold_time The required hold time for the element. Used only for clocked elements.

marked A boolean field used during the D algorithm. If the field is *true*, any attempt to drive a fault through the gate is guaranteed to fail.

level The gate's level in the circuit. Equals the longest path to the gate from any primary input.

Before defining the edge type a subtype is needed

```
typedef struct t_val
   { int time ;
     int value ;
   } time_value ;
```

This structures holds pairs of time and value, indicating that at time *time* the line was assigned the value *value*.

The edge type is

```
typedef struct edge
   { char user_name[8] ;
     int source ;
     int destination ;
     int value ;
     int fault ;
     time_value history[max_timesteps] ;
   } line ;
```

The fields of the structure are :

user_name A 7 character user name for the line.

source A number representing the source gate of the line.

destination A number representing the destination gate of the line.

value The logic value of the line.

fault The fault value of the line.

history An array of all the changes in the line value.

2.1.3 Data structures

`gate gate_list[max_gate_number]`

The internal name of all circuit gates is an integer. The integer value is the index of the gate's structure in the *gate_list[]* array.

`int fan_table[max_gate_number][max_fan+1]`

A primary input *gate* has just one output line and no input lines. The internal name of the output line is stored in the *q* field of the gate structure. A primary output gate has just one input line and no output line. The input line is stored in the *q* field of the structure.

A buffer has just one input line but multiple output lines, while all other combinational gates have one output line but multiple input lines. The one input (output) line of the gate is stored in the *q* field of the gate structure. The multiple input (output) lines are stored in *fan_table[][]*. If the gate name is i, then column 0 of row i holds the fan count while the following columns hold the names of the fan lines, in the order in which they appear in the circuit description file.

For a clocked element there are at most two output lines, a *q* line and a *qnot* line. The *q* and *qnot* fields of the element structure hold the names of theses lines. If a line is not connected to one of these outputs, the corresponding field is -1. At least one output is connected to some line. For a D flip-flop, there can be up to four inputs, two of which are optional. The four lines are the *clock, reset, preset* and *D* lines, with *reset* and *preset* being the optional lines. This is the order in which they appear in the circuit description and the order in which they appear in the element's row of *fan_table[][]*, starting in column 0. If any of the optional lines is missing, a -1 appears in its place. The same applies to a JK flip-flop, except it can have five input line, two of which are optional. The lines are *clock, reset, preset, J* and *K*, with *reset* and *preset* being optional.

`line line_list[max_line_number]`

line_list[] stores the line structures for all circuit lines. The internal line name is its index in *line_list[]*. The numbers are given in the order in which the lines are encountered in the circuit description.

`int line_x_reference[x_ref_length]`

When a user name for a line is read, it must be converted to an internal name. This is done with a *Hash* function and *line_x_reference[]* is the *Hash* table.

`int clk_table[max_clocks][max_clock_gates]`

The circuit can have up to *max_clocks* different clocks and each clock can drive up to *max_clock_gates* different gates. For each clock, *clock_table[][]* stores the names of the elements driven by the clock.

`int clk_cntr[max_clocks]`

For each clock, *clk_cntr* stores the number of elements driven by that clock.

`char clk_names[max_clocks][8]`

For each clock, *clk_names[][]* stores its user name. The name can be at most seven characters long.

`int number_of_gates`

Stores the number of gates in the circuit.

`int number_of_lines`

Stores the number of lines in the circuit.

`int first_pi`

In *gate_list[]* there are three distinct blocks. The first is the block of all combinational and clocked elements. It is followed by a block of all primary inputs, which is followed by the block of all primary outputs. *first_pi* is the index, and name, of the first primary input gate.

`int first_po`

first_po is the index, and name, of the first primary output gate.

2.2 D algorithm variables

2.2.1 Constants

action constants :

imply 0

drive 0

justify 1

2.2.2 Data types

For the D algorithm three lists are maintained. The first is the list of gates on the d-frontier. The second is the list of line justifications of the j-frontier. The third list is a list of all implications, i.e. implied line values and justifications added to the j-frontier, implied by a decision. This list is maintained so that if backtracking occurs and the decision is cancelled, the state of the circuit, before the decision, can be reconstructed.

A d-frontier structure is of type

```
typedef struct d_rec
   { struct d_rec *prev ;
     int gate ;
     struct d_rec *next ;
   } d_front_rec ;
```

The fields of the structure are

prev A pointer to the previous structure on the list.

gate The internal gate name of the d-front gate.

next A pointer to the next structure on the list.

To represent a line on the j-frontier we use

```
typedef struct j_rec
   { struct j_rec *prev ;
     int line ;
     struct j_rec *next ;
   } j_front_rec ;
```

The fields of the structure are

prev A pointer to the previous structure on the list.

line The internal line name of the j-front line.

next T pointer to the next structure on the list.

The line structure, in *line_list[]*, holds the line value needing justification, so there is no reason to store it here also.

To represent an implication structure the following structure is used

```
typedef struct act_rec
   { struct act_rec *prev ;
     int site ;
     int action ;
     struct act_rec *next ;
   } imply_rec ;
```

The fields of the structure are

prev A pointer to the previous structure on the list.

site The internal name of line involved in the implied action.

action An integer having the value of either the constant *imply* or the constant *justify*. The value determines whether the implication was a line assignment (*imply*) or the addition of a line to the j-frontier (*justify*).

next A pointer to the next structure on the list.

2.2.3 Data structures

`imply_rec *imply_ptr`

For every decision made, there is a local variable pointing to the first element of the implication list. Every level of recursive calls to the D algorithm has its own implication list. The pointer *imply_ptr* points to the last structure in the implication list of the current recursion level.

`int d_front_count`

d_front_count counts the number of gates on the d-frontier. This does not include primary outputs with values of *d* or *dbar*.

`int d_out_count`

d_out_count counts the number of primary outputs with a value of *d* or *dbar*. A non-zero value indicates that a fault has been propagated to a primary output. If both this counter and *d_front_count* are zero then we must backtrack.

`int j_front_count`

j_front_count counts the number of lines on the j-frontier. When this counter is zero and *d_out_count* is greater than zero, a test vector has been found.

`d_front_rec *d_front_hdr, *d_front_last`

These pointers point to the first and last elements of the d-frontier list.

`j_front_rec *j_front_hdr, *j_front_last`

These pointers point to the first and last elements of the j-frontier list.

`int fault_gate`

This variable hold the internal name of the destination gate of the faulty line. It is used to ensure that when the fault is first-level excited (the non-stuck-at value is first level justified, i.e. backward implication is carried out as far back as possible) the gate output will not be set. If the gate output is set, the fault cannot be later driven through this gate and propagated to a primary output.

`int d_alg_error`

A boolean flag indicating a fatal error occurred and the algorithm can not continue. An appropriate message will be generated. One such error is a flip-flop in the circuit, since the D algorithm works only on combinational circuits.

`int step , trace`

Boolean flags indicating whether one of the two special running modes, *step* or *trace* is in effect. Both flags have default values of *false*.

2.3 Fault collapsing variables

2.3.1 Data types

During the fault collapsing process, several lists will be created and maintained. All lists will have the same element type :

```
typedef struct f_rec
   { struct f_rec *prev ;
     int fault ;
     int line ;
     struct f_rec *next ;
   } fault_rec ;
```

The fields of the structure are

prev A pointer to the previous element on the list.

fault A 0 or a 1 indicating whether the fault is a *sa-0* or *sa-1*.

line The internal name of the faulty line.

next A pointer to the next element on the list.

2.3.2 Data structures

`fault_rec *redundant_list`

This pointer points to the next available element in the *redundant* fault list (the last element, which has all its fields set to *null*). This is a list of faults for which our test generation algorithm failed to produce a test.

`fault_rec *redundant_hdr`

A pointer to the first element of the redundant list.

`fault_rec *fault_list`

This pointer points to the next available element in the *fault* list (the last element, which has all its fields set to *null*). This is a list of faults for which we try to generate tests.

`fault_rec *fault_hdr`

A pointer to the first element of the fault list.

`fault_rec *dominant_list`

This pointer points to the next available element in the *dominant* fault list (the last element, which has all its fields set to *null*). This is a list of faults which represent dominating equivalence classes.

`fault_rec *dominant_hdr`

A pointer to the first element of the dominant list.

`fault_rec *equivalence_reps`

This pointer points to the next available element in the *equivalence representatives* list (the last element, which has all its fields set to *null*). Each fault on this list belongs to a different equivalence class and we start constructing the class with this element.

`fault_rec *reps_hdr`

A pointer to the first element of the equivalence representative list.

`fault_rec *equivalence_list`

This pointer points to the next available element in the *equivalence representatives* list (the last element, which has all its fields set to *null*). All faults on this list belong to the same equivalence class and are waiting their turn to be processed, i.e. are their corresponding input faults in the equivalence class or do they belong to new equivalence classes.

`fault_rec *equivalence_hdr`

A pointer to the first element of the equivalence list.

`int equivalence_table[8][max_line_number]`

All equivalence-dominance relations between all stuck-at faults are stored in the equivalence table. Each column represents a line. Each integer entry represents a fault. An entry of $2i$ represents that line i is stuck at 0. An entry of $2i+1$ represents line i stuck at 1.

All faults in an equivalent class are arranged in a pointer loop, i.e. each fault points to another fault in the class and by going around the loop, the whole class is encountered.

Row 0 (1) contains a stuck at 0 (1) equivalence pointer.

Row 2 (3) contains the representative of the stuck at 0 (1) equivalence class.

Row 4 (5) contains a fault dominating the stuck at 0 (1) fault. If no such dominating fault exists, the entry is -1.

If line i stuck at 0 (1) is an equivalence class representative, then row 6 (7) contains the number of faults dominated by the stuck at 0 (1) fault. If it is not a representative, the entry is -1. When a test for a dominated fault is found, the count of the dominating representative, through the whole dominating chain, is changed to -1. When a test for a dominated fault can not be generated, the count of the dominating representative is decreased by 1. If the count reaches 0, the fault is added to the list of fault for whom we must find a test vector (since no test was found for all the dominated faults).

`int active`

Represents the current fault begin processed in the fault collapsing process.

`int first`

The first fault of an equivalence class. Also the class representative.

`int dominant`

A boolean flag indicating whether the equivalence class dominates any other class. Initiated to false when the processing of the class begins. If the class does not dominate another, the class representative is added to the representative list. If the class does dominate some other class, the representative is placed on the dominant list.

2.4 Good Circuit Simulator variables

2.4.1 Constants

action constants :

init 0

add 1

pick 2

2.4.2 Data types

During event-driven simulation, three lists are maintained. The first is an event-list which stores the list of events to be processed at each timestep. The other two are sets - the **activated set** which is a set of gates activated by the current event and the **scheduled events set** which is a set of events scheduled by the current event.

An event structure is of type

```
typedef struct event_node
{ int line;
  int value;
  struct event_node *next;
} event;
```

The fields of the structure are

line The internal line name on which the event occurs

value The value which the line takes as a result of the event

next A pointer to the next structure on the event-list

Each element of the Activated Set is of type

```
typedef struct gate_element
{ int gate;
  struct gate_element *next;
} activated_gate;
```

The fields of the structure are

gate The internal gate name of the gate activated by the event

next A pointer to the next structure in the list

Each element of the Scheduled Events' set is of type

```
typedef struct event_element
{ int line;
  int value;
  int time;
  struct event_element *next;
} scheduled_event;
```

The fields of the structure are

line The internal line name of the line on which the event is scheduled

value The value the line is to take

time The time at which the event is scheduled to take place

next A pointer to the next structure in the list

2.4.3 Data structures

`int time`

time counts the simulation time. It is incremented every timestep.

`int simulation_length`

This variable holds the total simulation time.

`int number_of_events, number_gates_act`

These variables hold the total number of events simulated and the number of gates activated during the simulation run, respectively. The variables are for statistical data of the simulation run.

`event *next_event, *present_event`

present_event is a pointer to the current event and *next_event* is a pointer to the next event to be processed.

`event *event_list[max_timesteps]`

Each element of the array *event_list* is a linked list of events corresponding to that timestep. During simulation, for every timestep, time is advanced and events are picked up from the event list to be processed. Also newly scheduled events are places in the list corresponding to the scheduled time.

`activated_gate *activated_set`

This is a pointer to the first element of the activated set of gates.

`scheduled_event *scheduled_event_set`

This is a pointer to the first element of the set of new events to be scheduled.

`char step_flag, act_flag, event_flag, list_flag, line_flag`

These are flags indicating user responses to questions asked for display of information during simulation in the **step** mode. *act_flag* represents the activated set, *event_flag* represents current event, *list_flag* represents the event list and *line_flag* represents line values to be displayed. Depending upon the user responses these flags are set and information displayed.

`int observe_lines[max_line_number/5]`

This array holds the internal line names of the lines which the user wants to observe during the simulation run. The value of the lines in this array is printed out after every event is processed.

2.5 Fault simulator variables

2.5.1 Constants

INTSIZE 32

MAXINTS 256

ALLONES 1

NINTS (max_line_number + INTSIZE)/INTSIZE

2.5.2 Data types

The deductive fault simulation algorithm drops faults after simualtion of a test vector. An array of structures *d_fault* is used to keep a track of the faults to be dropped.

```
typedef struct fault
{  int line;
   int value;
}d_fault;
```

The fields of the structure are

line Internal line name of the line

value The stuck_at value of the line which is detected as a fault by the current test vector.

2.5.3 Data structures

```
int sim_f_list0[MAXINTS], sim_f_list1[MAXINTS]
```

sim_f_list0 is an array represnting the *sa_0* faults to be simulated. It is an array of integers and each *bit* of an integer corresponds to a line. If the bit is set, that line *sa_0* is to be simulated. Similarly, *sim_f_list1* is for *sa_1* faults.

```
int number_detected_faults, number_test_vectors
```

These variables are used for counting the number of faults detected and the total number of test vectors simulated. Statistical information like fault coverage can be calculated from these variables.

`int cntrl_value`

While finding the output fault list of every gate, this variable is set to 1, if an input line has a controlling value, else it is set to 0. Input fault list manipulation is then done depending upon this value.

`FILE *vector_file`

A pointer to the file containing the test vectors.

`d_fault faults_to_drop[max_line_number]`

An array of structures representing the faults to be dropped after simulating a test vector.

`int ckt_depth`

This variable stores the depth of the circuit after the circuit is levelized.

3 The circuit parser

The parser is written in accordance with *Lex* and *Yacc* format. Like most other things, the parser module (*parser.y*) can be divided into three submodules, the *Lex* specification file (lex.l), the *Yacc* specification file and the resulting parser source code files (*lex.yy.c, y.tab.c*). The source code files are generated automatically and our interest is only in the two specification files.

The semantic attribute (*yylval*) of the tokens and variables is an integer. The other variables shared by the *Lex* and *Yacc* routines are

`int line_num`

An integer that keeps count of the line numbers in the circuit description file. It is initialized to 1 and incremented when a newline character ($\backslash n$) is read.

`char names[2*max_fan][8]`

Every line in the circuit description file defines a different gate. Most of the information about the gate can be stored only when the entire line has been read. Some of this information is the names of the input and output lines. These are placed, by *Lex*, in the string *yytext*, and every time a new string is read, it writes over the previous string. Hence, we must store these names somewhere else. *names[][]* is used to store all the line names associated with a gate description in the circuit description file.

`int name`

A counter of the number of names stored in *names[][]*. It is also a pointer to the next available row in *names[][]*.

3.1 The lexical analyzer

The lexical analyzer has three major responsibilities. The first responsibility is, of course, the identification of tokens and passing them on to the parser. Its second responsibility is to catch certain syntax errors. The third task is a cleanup and maintenance job.

In identifying tokens, the lexical analyzer allows each keywords to be entered in either upper or lower case. A keyword can not, however, be written in both lower and upper case.

All line and gate user names are allowed up to seven characters. If a longer name is used, it is truncated, leaving only the first seven. All messages concerning that gate or line will include only the truncated name. As long as all seven character prefixes are distinct, the system will not care. If gate names share a seven character prefix, system messages concerning those gates will be ambiguous (although the user name for these gate will appear to be the same, the internal names will be different). If line names share a seven character prefix, the system will generate some kind of error message during the parsing procedure and the values placed in the data structures might not be valid.

When specifying delay parameters for a gate, the order of these parameters is fixed. The order is *rise, fall, setup* and *hold.* Some parameters may be omitted, but their relative order can not be changed. For every parameter read, a flag is set to true and the flags of all parameters that follow it are checked to be false.

When describing the input lines of a flip-flop the *preset* and *reset* lines are optional. If both appear, the preset line must come before the reset line. Boolean flags are used to check these requirements.

When describing the output lines of a flip-flop, one of either the q or the *qnot* lines must be specified. Both can also be used. When both are used, the q line must come before the *qnot* line. These requirements are also checked by flags.

The cleanup jobs include the identification of *C*-like comments in the circuit description file, ignoring white spaces and tab characters. The maintenance jobs include keeping track of the line numbers and catching unexpected characters in the stream, i.e. characters not in a comment that are not part of any token. When a newline ($\backslash n$) character is read, besides incrementing the *line_num* counter, all boolean flags are set to *false.*

The lexical analyzer will generate a message whenever a name is truncated, an ordering syntax error is found or an unexpected character is encountered.

3.2 The parser

The parser, like most other functional elements in the system, has three major responsibilities. The first and foremost responsibility of the parser is to read the circuit description file and convert the text description of the circuit to a description based on our data structures. These actions include

1. Sets the fields of all the elements of *gate_list[].*
2. Sets the fields of all the elements of of *line_list[]*, except line values.

3. Sets the elements of *line_x_reference[]*.

4. Sets the entries in *fan_table[][]*.

5. Sets the counters *number_of_gates* and *number_of_lines*.

6. Sets the pointers *first_pi* and *first_po*.

7. Creates an initial list of representatives of fault equivalence classes.

The second task the parser has is that of a semantic checker. The semantic checks that are carried out are

1. No two lines have the same user name.

2. Every line has a source and destination, i.e. there are no dangling lines.

3. The number of input lines and output lines of a gate are in accordance with the gate definition.

4. The number of gates and lines in the circuit does not exceed the allowed maximum.

5. The clocks driving the different flip-flops have all been declared.

The third and obvious task is a syntax checker for the circuit description file.

3.2.1 Parser variables

Besides the variables that are shared with the lexical analyzer, the parser uses the following variables.

`int parse_err`

A boolean flag that indicates whether a syntax or semantic error occurred during parsing process.

`int lines[2*max_fan]`

An array corresponding to the array of user names, *names[][]*. While *names[][]* holds user names associated with a line in the circuit description file, *lines[]* holds the internal names for those lines. The running pointer for *lines[]* is the variable i, which is reset to zero after a gate has been derived.

`int rise_val, fall_val, setup_val, hold_val`

These variables are assigned either the user specified delay parameters or, when no parameters are specified, -1. When the whole gate is derived, if a delay parameter is -1, the gate is given the default delay value of the gate type. If the delay parameter is not -1, the gate is given that value as its delay value.

```
fault_rec *fp
```

The parser constructs an initial list of representatives of equivalent fault classes. The pointer **fp* points to the last element of the list, which is empty and ready to accept a class representative.

```
int number_of_clocks
```

A counter for the number of clocks in the circuit.

3.2.2 Procedures and functions

```
int add_line(user_name)
char *user_name ;
```

This function receives a pointer to a line user name and adds that name to the *Hash* table. It returns the internal name for the line. If the name was already in the table, the function returns the internal name. When we say that a name is in the table, we mean the internal name corresponding to the user name. To check whether an internal name matches a user name, we check the *user_name* field in the line structure pointed to by the internal name.

```
add_to_gate_list(gate,type)
int gate, type ;
```

This procedure receives two parameters, the internal name of a gate and an integer equal to the constant corresponding to the gate type. The procedure first sets the fields of the gate structure in *gate_list[]*. It then sets the gate entries in fan_table and sets the source or destination fields for its output and input lines. Finally it resets to zero the running indices i and *name* of the arrays *lines[]* and *names[][]*.

The algorithm is

1. Set the fields of the gate entry in *gate_list[]*.
   (a) Set *type, q* (and *qnot*).
   (b) For those delay parameters not equal to -1, set the corresponding delay field to the parameter value. For those delay parameters equal to -1, set the corresponding delay fields to the default values of the gate type.

2. Using i and *lines[]*, set the gate row in *fan_table[][]* and for each line set its destination (source) field to *gate*.
3. Reset *name* and i to zero.

```
dangling_lines()
```

After parsing is completed, this procedure goes over all the source and destination fields in *line_list[]* and verifies that none of the entries are equal to or greater than *max_gate_number*.

```
int find_gate_name(s)
char *s ;
```

This function receives a pointer to a user name and returns the internal name of the gate. The only time a conversion is needed from a user gate name to an internal gate name is when the name is read from the circuit description file. The internal name given to the gate is the current value of *number_of_gates*, the gate counter for the circuit. If this value is greater than *max_gate_number*, an error message is generated. Otherwise, the user name is copied into the *user_name[]* field of the gate structure pointed to by *number_of_gates*, which is then incremented.

```
gen_err(err_num,site_name,line_num)
int err_num ;
char *site_name ;
int line_num ;
```

The procedure parameters are an error message number, a pointer to a string with the name of the site (gate or line) in error and a line number in the circuit description file. If the line is not known, the number is -1. The appropriate error message, with the corresponding site name and line number is generated.

```
gen_warn(warn_num,site_name,line_num)
int warn_num ;
char *site_name ;
int line_num ;
```

The procedure parameters are a warning message number, a pointer to a string with the name of the site (gate or line) involved and a line number in the circuit description file. If the line is not known, the number is -1. The appropriate warning, with the corresponding site name and line number is generated.

```
int get_line(user_name)
char *user_name ;
```

The function receives a pointer to a line user name and returns the internal name for the line. If the specified name is not in the table, an error message is generated, stating that the name can not be found.

```
int hash_name(user_name)
char *user_name ;
```

This function returns the initial *Hash* index for the user name. The *Hash* function used is the integer whose base 256 representation is the user name, taken modulo the length of the *Hash* table ($2 * max\_line\_number - 1$).

```
init()
```

This procedure initializes certain fields in the global data structures before starting the parsing process. The initialization steps are

1. In *line_list[]*, the source and destination fields are set to *max_gate_number*. After parsing is completed, these fields are checked. If any of them has a value equal to *max_gate_number* then the line does not have either a source or destination, depending on the field.
2. In *line_list[]* the user name is set to the empty string.
3. The entries in the *Hash* table, *line_x_reference*, are all set to -1.
4. The equivalent fault classes lists and pointers are initialized.
5. All clock counters are set to zero and all clock names to the empty string.

```
int parse(circuit)
FILE *circuit ;
```

This is the function one should use when he wants to read a circuit description file. The parameter to the function is a *FILE* pointer to the circuit description file. The function returns the constant *success* or the constant *fail*, depending on whether an error was detected during the parsing process.

```
print_tables()
```

This procedure prints the global structures *gate_list[], line_list[], fan_table[][], clock_table[][], line_x_reference[]* and the initial list of fault equivalence classes representatives.

```
yyerror(s)
char *s ;
```

A *Yacc* routine that generates the message pointed to by s. Can be overridden by a (*Yacc*) user specified routine. Our version sets the *parse_err* flag to true and in addition to the message in s, generates the line number of the circuit description file that is currently read. This gives a fair idea as to the location of the error.

3.2.3 Changing the parser

Any change to the parser will involve a combination of changes to the lexical analyzer, the input grammar and the semantic routines (it's that magic number three again). As an example, suppose we want to add a new gate type, *newtyp*. First we must recognize this new keyword, and this involves an addition to the lexical analyzer. Next, we must add a derivation rule for this new type, and this requires a change in the grammar. Opon deriving a gate of this type, we need to tell the parser what to do, and so we must add the appropriate semantic routines.

3.2.4 Using the parser

To use the parser, the user object files must be linked with the object files *y.tab.o, lex.yy.o* and *fc_lists.o* . These files can be found in the parser directory (*parser*). The user must also include the files *global.h* and *global.g* found in the main directory and the files *fc.h* and *fc.g* found in the fault collapsing directory (*fc*). An example file, *parse.c*, can be found in the parser directory (*parser*).

3.2.5 An exapmle program

The following example program is in the file *parse.c* in the *parser* directory.

```
/*************************************************************************
  A program that wants to use the parser must :

  1. include "global.g"
  2. open the circuit file in read mode
  3. call parse with 1 parameter :
      i. the file pointer to the circuit file
  4. if parse returns success, there is no parsing error,
     if parse returns fail there is a syntax or semantic error in the circuit
     file.
  5. link with /home/poisson3/ee658/parser/y.tab.o,
               /home/poisson3/ee658/parser/lex.yy.o,
               /home/poisson3/ee658/parser/fc_lists.o

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

#include "/home/poisson3/ee658/global.h"
#include "/home/poisson3/ee658/global.g"

main(argc,argv)
int argc ;
char *argv[] ;
```

```
{ FILE *circuit ;
  int ok ;

  circuit = fopen(argv[1],"r") ;
  ok = parse(circuit) ;
  if (ok == success) print_tables() ;
}
```

The program is called with the circuit description file as a parameter. It first open the description file for reading and then calls the parsing procedure *parse()*. If *parse()* returns *success*, it calls *print_tables()*, which prints the global structures *gate_list[], line_list[], fan_table[][]* and *line_x_reference[]*.

4 Test generation

The problem of test generation is divided into three subproblems. The first is fault collapsing, to obtain a minimal test set. During the fault collapsing process all dominating relations between fault classes are found and a hierarchy is maintained. The test generation process begins with a minimal set containing only dominated classes and a dominating class is added only whena test for all its dominated faults cannot be found. Maintaining these relations and keeping track of all the book keeping involved is the responsibility of the second module. The third module is the test generation module, which, when given a fault, tries to find a test vector for it. This module is an implementation of the D algorithm.

4.1 Fault collapsing

First the fault collapsing process is described - how equivalence classes are built and how dominating relations are established. Then the model used for the dominating hierarchy and the way this model is implemented is described. Finally the algorithm used for fault collapsing and a description of all the functions involved is presented.

4.1.1 Building the equivalence hierarchy

The fault collapsing process is based on the local equivalence and dominating relations between input and output faults of a gate. For the five gate types *AND, OR, INV, NAND, NOR,* these relations are easily determined. Basically, if c denotes the controlling value of the gate and i denotes the inversion value of the gate, then all input faults stuck at c are equivalent to the output stuck at $c \oplus i$ and an input stuck at $\overline{c}$ is dominated by the output stuck at $\overline{c \oplus i}$. For a *buffer*, which models a fanout point, there is no equivalence-dominance relation between stem faults and branch faults. The same applies for an *XOR* gate, there is no equivalence-dominance relation between an input fault and an output fault. Therefore, the *buffer* and *XOR* outputs will be called *stop lines* and the *buffer* and *XOR* inputs will be called *start lines*.

This leads us to the following observations

1. The process of fault collapsing is an inherent *backward* procedure on the circuit graph, i.e. you begin with the primary outputs and start lines, and advance, backwards, to the primary inputs.

2. As long as the output fault is equivalent to all corresponding input faults, you add the input faults to the list and continue the process with these faults.

3. When the output fault dominates an input fault, the input fault is the first element of a new equivalence class. This new class will not include any other fault on the input lines of the gate.

4. An equivalence class has just one dominating class, while it can dominate more than one class. The second part is trivial. The first follows from the construction process. Faults are added to the class only when they are input faults equivalent to the output fault. Since a gate has just one output, there is no fault that can dominate these additional faults.

5. When a stop line is reached, no additional faults can be added to the class and no dominance relation exists between the fault and the corresponding start line faults.

As a result of these observations, the collapsing strategy is to start with an initial representative list of all primary output faults and start line faults. These are all representatives of different equivalence classes. This initial list is constructed by the parser. For each representative, a backward search for equivalent faults is initiated. When a dominated fault is found, since it represents a new equivalence class, it is added to the list of representatives. The process ends when the representative list is emptied.

4.1.2 Modeling and representing the equivalence hierarchy

To represent the equivalence hierarchy a *directed acyclic graph* is used. The nodes of the graph are the equivalence classes and an edge exists from node i to node j if and only if class i is dominated by class j. Thus, in the beginning of the test generation process, the *in-degree* of a node is the number of classes it dominates. The *out-degree* of each node is one, since a class can be dominated by just one other class. The initial fault list consists of all classes whose *in-degree* is zero. When a test is found for a class, all the faults in the class, and all the faults in all the dominating classes of that class, can be tested for, thus, the original class and all the nodes reachable from the original class can be erased from the graph. When a test for a class cannot be generated, the class node and its outgoing edge from the graph is erased. This decreases by one the *in-degree* of the dominating class. When the *in-degree* of a dominating class is zero, it means that a test for all its dominated classes cannot be generated, hence, to catch the class, an attempt to generate a test for it must be made. Thus, the list of faults always consists of all nodes with an *in-degree* of zero.

In implementing this model care must be taken for

1. Representing all equivalence classes.

2. Representing all dominance relations.

3. Be able to eliminate classes from being considered for test generation.

4. For each dominating class, keep track of the number of dominated classes still begin considered for (possible) test generation.

Only one table is used to represent all the information needed. Each column of the table represents a line, where, obviously, column i represents line i. The table has eight rows.

Row 0 (1) contains a name of a fault that is equivalent to line i being stuck at zero (one). Thus, each equivalent class is represented as a cyclic list. Starting at one fault and following the pointers will carry us through all the equivalent faults of the initial fault.

Row 2 (3) contains the name of the fault that is the representative of the class to which line i stuck at zero (one) belongs to.

If line i stuck at zero (one) is the representative of its equivalent class, then row 4 (5) contains the name of the fault that dominates the equivalence class. Otherwise, this field is -1. If there is no dominating fault for the class then this entry is -1.

Row 6 (7) is the count of the number of dominated fault classes still (possibly) considered for test generation. This field is only valid if line i stuck at zero (one) is the representative of its class. When a test can not be generated for a dominated fault, this field is decreased. When a test has been found for the class (either by test generation or by generating a test for a dominated fault) the field is changed to -1 and we traverse the path reachable from the node, changing all counters to -1 until either the path ends or a negative counter is reached (meaning this class is already caught by some other vector or vectors).

4.1.3 The fault collapsing algorithm

The algorithm maintains four lists. The first, *equivalence_reps*, is a list of equivalence class representatives. The second, *equivalence_list* is a list of equivalent faults that have not been processed, i.e. tested for equivalence or dominance with their corresponding input faults or for being stop lines. The third, *fault_list*, is a list of representatives for non-dominating fault classes, i.e. classes in the initial minimal fault list. The fourth list, *dominant_list*, is a list of all the dominating classes.

The algorithm also has two special fault statuses. *first* is the first fault in a class which is the class representative. When processing a class, *active* is the fault that is currently processed. A boolean flag, *dominant*, is checked at the end of the class processing, to determine whether it is a dominating class or not. Other variables used are f, the fault type, *site*, the fault site, *source*, the source gate of the fault site, c and i, the controlling and inversion values of the source gate.

To represent faults as integers we use the following formula. Line i stuck at zero is represented as $2i$ while line i stuck at one is $2i + 1$.

The algorithm goes as follows

1. While the *equivalence_reps* list is not empty

   (a) $active = -1$, $dominant = false$

(b) Choose a fault from *equivalence_reps*

   i. Add the fault to *equivalence_list*

   ii. Delete the fault from *equivalence_reps*

   iii. If the fault is stuck at f and the line is j, $first = 2j + f$.

   iv. The representative of line j stuck at f is *first*, the stuck at f dominated count of line j is 0.

(c) While *equivalence_list* is not empty

   i. Choose a fault from *equivalence_list*.

   ii. If the fault is line j stuck at f

      A. $equivalence\_table[f][j] = active$

      B. $active = 2j + f$

      C. $equivalence\_table[4 + f][j] = first$

   iii. If line j is a primary input, a buffer output or a *XOR* output, do nothing.

   iv. If line j is an inverter output, add the inverter input stuck at $1 - f$ to *equivalence_list*.

   v. If $f = c \oplus i$, add all input lines stuck at c to *equivalence_list*

   vi. If $f = \overline{c \oplus i}$

      A. Add all input lines stuck at $\overline{c}$ to *equivalence_reps*.

      B. $dominant = true$.

      C. For each input line k, $equivalence\_table[2 + \bar{c}][k] = active$ (meaning the current equivalence class dominates the equivalence classes of the input lines stuck at $\bar{c}$).

      D. $f = first \pmod 2$, $line = (first - f)/2$, $equivalence\_table[6 + f][line] += number\_of\_inputs$.

(d) $f = first \pmod 2$, $line = (first - f)/2$, $equivalence\_table[f][line] = active$.

(e) If $dominant == true$ add *first* to *dominant_list*, else add *first* to *fault_list*.

4.1.4 The fault collapsing functions

```
add_to_dom_list(fault,line)
int fault , line ;
```

Adds a new element to *dominant_list*. *fault* is the fault type, *line* is the fault site.

```
add_to_eq_list(fault,line)
int fault , line ;
```

Adds a new element to *equivalence_list*. *fault* is the fault type, *line* is the fault site.

```
add_to_eq_reps(fault,line)
int fault , line ;
```

Adds a new element to *equivalence_reps*. *fault* is the fault type, *line* is the fault site.

```
add_to_f_list(fault,line)
int fault , line ;
```

Adds a new element to *fault_list*. *fault* is the fault type, *line* is the fault site.

```
fc()
```

This function constructs the equivalence table and the minimal fault list. It assumes that the parser has already created a fault list which includes all primary outputs and start lines. The function follows the above fault collapsing algorithm to the letter.

```
fc_warn(num,site)
int num , site ;
```

Prints warning number *num* for site *site*. Currently there is only one warning message saying that fault collapsing over gates of the type of gate *site* is not implemented.

```
init_fc()
```

Initiates the lists (creates the first elements, sets the header pointer and current pointer to point to that element) *equivalence_list, equivalence_reps, dominant_list* and *fault_list*.

```
print_eq_tab()
```

This function is a useful debugging tool for *fc()*. It prints the equivalence table of the circuit, transposed, i.e. it prints rows as columns and columns as rows.

```
print_fc()
```

This function prints the equivalence classes in the fault list followed by the equivalent classes in the dominant list.

4.1.5 An example program

The following program is in the file *collapse.c* in the *fc* directory.

```
#include <stdio.h>

#include "global.h"
#include "global.g"

main(argc,argv)
int argc ;
char *argv[] ;
{ FILE *circuit ;
  int ok ;

  circuit = fopen(argv[1],"r") ;
  ok = parse(circuit) ;
  if (ok == success)
    { fc() ;
      print_eq_tab() ;
      print_fc() ;
    }
}
```

The program is called with the circuit description file as a parameter. It opens the description file in read mode and calls *parse()* to parse the circuit. If parsing is successful, it calls *fc()* to perform the fault collapsing and then it calls two output functions. The first, *print_eq_table()*, prints the equivalence table. The second, *print_fc()*, prints the equivalence classes. It first prints the classes on the fault list and then the classes on the dominant list.

4.2 Creating a test set

When creating a test set, we assume that fault collapsing has already been done and the equivalence table and fault list are ready. All that is left is to start taking faults off the fault list and trying to generate a test for it. If test generation is successful, the equivalence class, and all generations of dominating classes are considered tested, and the dominating count entries in the equivalence table must be changed accordingly. If test generation fails, the class is added to a list of redundant faults and the dominating count of the first generation dominating class is decreased. If this count reaches zero, the class is added to the fault list. The list of redundant faults is either a list of true redundant faults, if there are no resource limits (e.g. time, number of backtracks) on the algorithm, or, if there are resource limits, pseudo redundant faults, i.e. faults the test generation algorithm was not able to catch within the required limits.

The algorithm used is as follows

While the *fault_list* is not empty

1. Take a fault off the list.
2. Denoting the fault site as *site* and the stuck at value as f, call *gen_test(site, f)*.
3. If *gen_test* returned *success*
   (a) While *site* stuck at f is dominated
      i. $site = dominating\_site$, $f = dominating\_f$.
      ii. Change dominating count of *site* stuck at f to -1.
4. *gen_test* returned *fail*
   (a) Add *site* stuck at f to the redundant list.
   (b) If there is a dominating fault, decrement the dominating count of its representative.
   (c) If the result is zero, add this dominating class representative to the fault list.

4.2.1 Test set functions

```
add_to_red_list(fault,line)
int fault , line ;
```

Adds line *line* stuck at *fault* to the redundant list.

```
caught(site,fault)
int site , fault ;
```

The function is called by fault_sim() when it finds that line *site* stuck at *fault* is also caught by the simulated input vector. It finds the representative of the class the fault belongs to and proceeds to take it off the fault list (if it is on the list) and mark all dominating classes as also caught (by setting their dominating count to -1).

Function steps :

1. Find the representative fault of the equivalence class of line *site* stuck at *fault*.
2. If the representative is on the fault list, take it off.
3. While the representative is dominated
   (a) representative = dominating class representative
   (b) set dominating count to -1

```
int gen_test(site,flt)
int site , flt ;
```

The parameters to this function are an internal line name (*site*) and a stuck at value (*flt*). The function initializes the D algorithm variables, creates the implied actions list, does first-level excitation of the fault and implies the fault through the destination gate of the faulty line. If all this succeeds, it calls the D algorithm.

The function follows the following algorithm

1. Call *init_search()*.
2. Create the first element of the implied action list and set *imply_ptr* to point to it (although we will never use this list for backtracking, there is nowhere to backtrack to, subsequent functions use the list, hence we must provide it).
3. Set *fault_gate* to the destination gate of *site* (this is necessary to prevent the output of the gate from being set during first-level justification).
4. Call *imply_and_check(site,1-flt,justify)* to first-level justify (excite) the fault.
5. If *imply_and_check* returned *success*
   (a) Decrement the *logic_count[1-flt]* field of the destination gate.
   (b) Increment the *logic_count[x]* field of the destination gate.
   (c) Set *fault_gate* to *number_of_gates*.
   (d) Call *imply_and_check(site,d+flt,imply)* to imply the fault value through the destination gate of the fault site.
   (e) If *imply_and_check* returned *success* call *d_alg()* and return its returned value, else return an error message (first drive failed).
6. Else, return an error message (first level justification failed).

`gen_tests_init()`

Creates the first element in the d-frontier and j-frontier list and sets the header and current pointers of these lists (*d_front_hdr, d_front_last, j_front_hdr, j_front_last*) to point to their respective element.

`init_red_list()`

Creates the first element in the redundant fault list with null fields. It then sets the header pointer, *redundant_hdr*, and current pointer *redundant_list*, to point to this element.

`init_search()`

Initializes all the data structures and fields associated with the D algorithm. This function is called before beginning a search for a test vector with the D algorithm.

`gen_sim()`

The function assumes fault collapsing has been done. It picks a fault from the fault list and tries to generate a test for it. If a test is found, it carries out fault simulation for the produced test vector. At the end of the fault simulation process, all faults that are caught by the test are taken off the fault list and all the dominating calsses of these caught classes have their dominating count set to -1.

`test_set()`

This function first calls *init_red_list()* and *gen_tests_init()*. These functions initialize the redundant fault list and d-frontier and j-frontier lists used during the D algorithm. After these initializations, the function follows the above algorithm.

4.2.2 An example program

The example program can be found in the file *gen_test_set.c* in the *test_set* directory.

```
#include <stdio.h>

#include "/home/poisson3/ee658/global.h"
#include "/home/poisson3/ee658/global.g"

main(argc,argv)
int argc ;
char *argv[] ;
{ FILE *circuit ;
  int ok ;

  circuit = fopen(argv[1],"r") ;
  ok = parse(circuit) ;
  if (ok == success)
    { fc() ;
      test_set() ;
    }
}
```

The program is called with the circuit description file as a parameter. It opens the file and parses the circuit. If parsing succeeds, it calls *fc()*, to create the fault classes and equivalence table, and then calls *test_set* to generate a test set for all non-redundant faults.

4.3 The test generation algorithm

The test generation algorithm and its implementation is the heart of the test generation module. The algorithm used to generate a test for a given fault is the D algorithm. In this section, first certain aspects of the design and implementation are highlighted and a full description of each function used in the algorithm is given.

4.3.1 Design and strategy

In any implementation of the D algorithm, the following points must be cared for

1. Maintaining a list of all gates which have a fault on their input and their output value is not yet set. This list is known as the d-frontier.

2. Maintain a list of all lines whose value is not in conflict, but not yet determined, by the input values to their source gate. This list is known as the j-frontier.

3. For every decision made, keep track of all the changes in circuit line values that resulted from that decision (a decision is the assignment of a value to a line and the changes are the forward and backward implications of that assignment). In case of backtracking, the decision is cancelled and the changes that resulted from the decision must also be cancelled.

4. Keep track of the number of primary outputs with faulty values.

From the above considerations, a number of functions must be written with the following goals

1. List manipulation routines.

2. Selection functions for choosing the next action, whether it will be a line justification or an error propagation.

3. Imply and check routines for forward and backward implications. These routines carry the implication process as far as possible while checking that they do not conflict with unjustified line values.

4. Line justification routines. For every line needing justification, we must either set all free input lines to the non-controlling input value or set one input to a controlling value. In the first case there is no decision involved whereas in the second case we must choose which free input to set.

5. A *cancelling* routine that cancels all line assignments due to a cancelled decision.

A natural way to implement the D algorithm is by recursion. The recursion however uses both global variables, accessible to all recursion levels, and local variables, which belong only to one specific level. This enables the program not to make multiple copies of data structures for each call, thus saving a lot of memory space and context switch time, but it also makes cancelling decisions and restoring states more difficult, because it is not done automatically by the recursion mechanism.

The global structures are, besides the circuit structures (*gate_list, line_list, fan_table, ect.)*, the d-frontier and j-frontier lists with their header and current pointers and counters and the counter for the number of primary outputs with faulty values.

The most important local structure is the implied actions list and its header and current pointers. This list contains all the line assignments and all the line justifications that resulted from a line assignment decision. When this decision is cancelled, all line assignments must be cancelled (i.e. line values set back to x) and the resulting line justifications must be taken off the j-frontier.

An important aspect of every search procedure is pruning the search space as much as possible. One pruning method is implemented here. Assume the circuit is in a certain state, s, and we decide to drive a fault through some gate. At some point the recursive call we made following that decision returns with a *fail* value. This means that in state s, driving a fault through our gate can not produce a test vector. We must cancel this drive and choose another action, whether a drive or a line justification. In subsequent search stages, we might reach a point where the fault at the input of our gate is propagated to the output. Since we know that in state s, or any state that includes state s, we can not find a test when the output of our gate is faulty, it is useless to pursue this search path and we can return a *fail* value immediately. To keep track of these gates, we use the *marked* field in the gate structure. If this field is *true*, any attempt to propagate a fault through this gate results in failure. The field is changed to *false* only when we backtrack from the level where the drive decision was first made. This backtrack will change some previous decision, hence will change the state s.

Before going into a detailed description of the functions and algorithms, a brief top-down overview of our implementation is presented.

When a call is made to the D algorithm it first checks its lists. If a fault has been propagated to a primary output and there are no more lines in need of justification, it returns *success*. If there are no faults at any primary output and the d-frontier is empty, it returns *fail*. If a special error flag is true, it returns *fail*. In all other cases, it must select an action. This action is either justifying the value of a line or driving a fault through a gate .

When justifying a line we distinguish between two cases. The first is when the source gate of the line is of type *AND, OR, INV, NAND* or *NOR* and the second is when the gate is of type *XOR*.

In the first case we first check whether the line value is $c \oplus i$ or $\overline{c \oplus i}$, where c is the controlling value and i is the inversion value of the gate. When the output is $c \oplus i$ we must select a free input and set it to c. When the output is $\overline{c \oplus i}$ we must set all free inputs to $\bar{c}$. In both cases we add all the assignments to the implied list and carry out backward implication for these assignments.

When the source gate is an *exclusive-OR* gate, since we allow only two-input *XOR* gates, there are only two ways to justify a line. A decision was made as to which of the two alternatives to try first. After setting these line values, backward implication is carried out.

When doing backward implication, if a line receives a values of $\overline{c \oplus i}$ (c and i are source gate dependent) all free input lines to the source gate are set to $\bar{c}$ and the backward implication continues with these lines. If a line receives a value of $c \oplus i$ we check how many free inputs there are. If there is only one, it is set to c and the process continues. If there are more than one, a decision has to be made as to which free input to set, hence the line is added to the j-frontier. A special case arises when the source gate is a *BUF, INV* or *XOR*. In case of a *buffer*, backward implication continues on the stem and forward implication is carried out on all other branches. In case of an *inverter*, the input line is assigned the complement of the output and backward implication continues. In case of an *exclusive-OR*, if one input is specified, backward implication continues with the free input. If both inputs are free, the output line is placed on the j-frontier. All line assignments and placements on the j-frontier are also recorded on the implication list.

If all backward implication returned *success*, a recursive call to the D algorithm is made. If it returns *success*, the calling level also returns *success*. If backward implication failed or if the recursive call returns *fail*, the decision is reversed, i.e. all implied actions are cancelled and the line assignment is complemented (with the appropriate backward implications carried out). If another decision can be made to justify the output, the process continues as before. If all options were tried and failed, the line is placed back on the j-frontier and backtracking occurs (*fail* is returned).

If the D algorithm decision was to drive a fault, we first set all free gate inputs to the non controlling value and carry out backward implications. If all backward implications return *success*, we imply the fault through the gate. If forward implication returned *success*, we make a recursive call to the D algorithm. If this call returns *success*, the current level returns *success*. If forward implication returned *fail* or if the recursive call returned *fail*, the gate is *marked* and a call is made to the D algorithm. If this call succeeds, the current level returns *success*. If the call fails, the gate is *unmarked* and we return *fail*.

During the *D drive*, the initial gate is taken off the d-frontier and all the gates, on whose inputs the drive ended with faulty values, are added to the front. The d-front counter and the faulty output counter are changed accordingly.

Whenever backtracking occurs, we must cancel actions. All the actions due to a decision were recorded in the implication list of the level in which the decision was made. Each element in the list is an action (imply or justify) and a line name. If the action is imply, the line value is changed to x. If the action is justify, the line is taken off the j-frontier (there is no more need to justify its value, since the assignment was cancelled). When cancelling a line assignment, if the source gate has erroneous inputs, then it is added to the d-frontier. If the cancelled assignment had an erroneous value and the destination gate has no other erroneous inputs, the gate is taken off the d-frontier. This will restore the state of the d-frontier to its state before the decision was made.

One last remark on the treatment of the j-frontier. A line is placed on the front when the inputs to its source gate do not imply its value. From the time the line was put on the front, to the time it is taken off for justification, forward implications may have already justified its value. Thus, before taking any action a check is made if the line value is implied by the source gate inputs. If so, the D algorithm is called again. If not, we proceed with the justification.

4.3.2 D algorithm functions

```
int add_to_d_front(gate)
int gate ;
```

If the gate is *marked*, the function returns *fail*. If the gate is a primary output, the function increments the *d_out_count* counter and returns *success*. Otherwise, the *d_front_count* counter is incremented, the gate is added to the d-frontier and the function returns *success*.

```
add_to_imply_list(line,val,action)
int line , val , action ;
```

The function adds a new element to the implication list, stating that action *action* was performed on line *line*. If the action is *imply*, the function changes the *value* field of the line structure in *line_list[]* and changes the *logic_count[]* of the destination gate in *gate_list[]*. If the action was *justify*, the function adds the line to the j-frontier.

```
add_to_j_front(site)
int site ;
```

The function adds line *site* to the j-frontier, placing it at the end of the list.

```
imply_rec choose_next_action()
```

The function returns a structure with two fields. The first is the *action* field, which takes the value of either *justify* or *drive*. The second is the *site* field which is an integer representing either a line or a gate, depending on whether the action is justify or drive. These fields are the decision for the next D algorithm action. The selection strategy is that as long as there is no fault at a primary output, we drive. Once a fault reaches a primary output, we start justifying. Whether we take an action from the d-frontier or from the j-frontier, it is always the element at the end of the list, thus it is assumed that any heuristic used to order the drives or justifications is implemented by the insertion routines to these lists.

```
int compute_val(gate)
int gate ;
```

The function receives the name of a gate and returns the output value implied by the input values. The computation is done by the count of each logic value at the input.

```
d_alg()
```

The D algorithm function call. It checks the conditions for immediate success or failure and if none of them are in effect, it initializes the implication list, selects an action and calls the function that carries out the selected action. The function steps are as follows

1. If $d\_out\_count > 0$ and $j\_front\_count == 0$, return *success*.
2. If $d\_front\_count + d\_out\_count == 0$, return *fail*.
3. If $d\_error\_flag == true$, return *fail*.
4. Initialize the implication list with the local pointer *imply_list* and the global pointer *imply_ptr* pointing to the first element.
5. Call *choose_next_action()* for the action and site of the next decision.
6. If the decision was *justify*, call *just_act(imply_list,action.site)* and return its returned value.
7. If the action was *drive*, call *drive_act(imply_list,action.site)* and return its returned value.

```
int drive_act(imply_list,gate)
imply_rec *imply_list ;
int gate ;
```

In the current circuit state, i.e. line values, the function tries to drive a fault through gate *gate*. If the drive succeeds, we make a recursive call to *d_alg()*. If this call returns *fail* or if the drive does not succeed (backward or forward implication fail), we cancel all implied values, take the gate off the d-frontier, and mark the gate. This mark will mean that in subsequent calls to *d_alg()*, further down in the recursion hierarchy, any search path driving a fault through *gate* is guaranteed failure, therefore there is no reason to continue the path. After marking the gate we make a call to *d_alg()*. If this call succeeds, we return *success*, if it fails, we put *gate* back on the d-frontier, unmark it and backtrack. The function steps are

1. According to the gate type, find $\bar{c}$ and i. If the gate is an *XOR* gate, arbitrarily set $\bar{c}$ and i to 0.
2. First-level-justify setting all free inputs of *gate* to $\bar{c}$.
3. If successful, call *d_alg()*.
4. If successful, imply the faulty output.
5. If successful, return *success*.
6. Undo all implications.
7. If the gate is of type *XOR*
   (a) Try the drive with $\bar{c}$ and i set to 1.
   (b) If successful, call *d_alg()*.
   (c) If successful, return *success*.
   (d) Undo all implications.
8. Take *gate* off the d-frontier and *mark* it.
9. Call *d_alg()*.
10. If successful, return *success*.
11. *Unmark* the gate, place back on the d-frontier and return *fail*.

```
int drive_fault(gate,dval,c,i)
int gate , dval , c , i ;
```

To drive a fault through a gate we have to set all free inputs to the non-controlling gate value (the c parameter) and first-level-justify these assignments. If they all succeed, we have to imply the resulting output ($dval \oplus i$). If this forwrd implication succeeds, we make a recursive call to *d_alg()* and return its result. The function steps are

1. Set all free inputs to the non-controlling value of the gate and first-level-justify.
2. If all backward implications succeed

(a) Remove the gate from the d-frontier.
(b) Forward imply the gate output.
(c) If successful
    i. Call *d_alg()*.
    ii. If successful, return *success*.

3. Return *fail*.

```
imply_and_check(line,val,action)
int line , val , action ;
```

This function does forward and backward implications and checks for conflicts that might result from these implications. If the *action* parameter is *imply*, we do forward implication. If it is *justify*, we do backward implication.

The function is called when a line value is either decided on or implied by existing values. First thing we do is add the value to the circuit and to the implication list. In forward implication we compute the value of the destination gate as a result of the assignment. If the value is x, we do nothing. If the current output is x and the new value is not x, we make a recursive call to *imply_and_check()* with the output line, the computed value and *imply* as the *action* parameter. If the new value and the current value are both not x and they disagree, we detect a conflict (this can only happen if the output line is on the j-frontier). If, during forward implication, faults are either propagated or cancelled, we update the d-frontier. When forward implication reaches a *buffer* input, we continue forward implication on all output lines. If forward implication reaches a gate where all input lines but one are set, and the output needs justification, the value of the free input is implied, hence we continue with backward implication on this line.

In backward implication we check whether we need a controlling or non-controlling value at the input. If we need a controlling value and if more than one input is free, a decision must be made as to which input to set to the controlling value. Therefore, we add the line to the j-frontier and we add the line to the implication list with the *action* parameter set to *justify*. If we need a non-controlling value, we check whether one of the inputs has a fault value. If not, we set all free inputs to the non-controlling value and continue with backward implication on these lines (if there's a fault on one of the input lines, setting the free inputs to the non-controlling value will propagate the fault, resulting in a conflict, since all justification values are either 0 or 1). If backward implication is done on a *buffer* branch, we continue with backward implication on the input line and forward implication on all other branches.

The procedure steps are as follows

1. Call *add_to_imply_list(line,val,imply)*.
2. If $action == justify$
   (a) If the source gate is a primary input, return *success*.
   (b) If the source gate is a *buffer*
       i. Continue backward implication on the input line.

ii. If successful, do forward implication on all other output lines, otherwise return *fail.*

iii. If all forward implication succeed, return *success*, otherwise, return *fail.*

(c) If the source gate is an *inverter*, continue with backward implication on the input line.

(d) If the source gate is of type *AND, OR, NAND* or *NOR*

i. If the required input is c and there is more than one free input line, add the line to the j-frontier and the implication list and return *success.*

ii. If the required input is c and there is just one free input, continue backward implication on this input.

iii. If there are faults on any of the input lines, return *fail.*

iv. Set all free inputs to $\bar{c}$ and continue with backward implications.

(e) If the source gate is of type *XOR*

i. If both inputs are free, add the line to the j-frontier and implication list and return *success.*

ii. If there is a fault on one of the inputs, return *fail.*

iii. Continue with backward implication on the one free input.

3. *action == imply*

(a) If the destination gate is a *primary output*, return *success.*

(b) If the destination gate is a *buffer*

i. Call *imply_and_check()* on all output branches.

ii. If all return *success*, return *success*, else return *fail.*

(c) Compute the new output value for the destination gate.

(d) If the value is x

i. If the new input was a fault, and there are no other faulty inputs, add the gate to the d-frontier.

ii. If the output is specified and there is only one free input left, continue with backward implication on the free input, and return the result.

iii. Return *success.*

(e) The current output is not x

i. If the current output and the implied output agree, return *success.*

ii. If the current output is a fault, it means that the output line is the fault site. Therefore, if the implied value is the excitation value, i.e. $1 - fault$, return *success.*

(f) If the current output is x

i. If the gate was on the d-frontier, take it off.

ii. Continue with forward implication on the output line.

(g) Return *fail* (both implied and current output values are not x and they disagree).

```
imply_free_input(gate)
int gate ;
```

This function is called to check whether the output of *gate* is specified and *gate* has just one free input. If both conditions do not exist, i.e. either the output is free or more than one input is free, the function returns *success*. If *gate* is not an *XOR* gate, than the free input is set to c and the function returns whatever value the backward implication returns. If *gate* is an *XOR* gate, then if it has a fault on an input, the function returns *fail*, otherwise it returns the value returned by the backward implication.

```
int just_act(imply_list,line)
imply_rec *imply_list ;
int line ;
```

The function receives a pointer to the first element of the implication list and the name of the line needing justification. It first checks whether the line value is already implied by the destination gate inputs. If so, it takes the line off the j-frontier, recursively calls *d_alg()* again and returns the returned value. If the line value is not implied by the gate inputs, it distinguishes between two cases. The first is when the destination gate type is *AND, OR, NAND* or *NOR*, the second is when the type is *XOR*. For the first case it calls *just_act_gate(imply_list,line,val)* and for the second case it calls *just_act_xor(imply_list,line,val)*, where *val* is the line value (if that value is *d* or *dbar*, val is 1 or 0 respectively). In both cases the function returns the value returned by the called function.

```
int just_act_gate(imply_list,line,val)
imply_rec *imply_list ;
int line , val ;
```

The function receives the pointer to the implied actions list, a line name and a value that needs justification for that line. It chooses which of the free inputs to set to the controlling value and does backward implications. If successful, it calls *d_alg()*. If *d_alg()* returns *success*, the function also returns *success*. If backward implication failed, or if *d_alg()* failed, all implications are cancelled, the line is placed back on the j-frontier, the chosen input line is complemented and backward implication is carried out. If successful, a recursive call to *d_alg()* is made. If *d_alg()* returns *success*, the function returns success. If either backward implication or *d_alg()* return *fail*, we cancel implications, set the chosen input line to x and return *fail*. The function steps are

1. Take the line off the j-frontier.
2. Depending on the gate type, find the controlling value c.
3. Choose a free input to set to c.

4. Set and first-level-justify this input.
5. If successful
   (a) Call *d_alg()*.
   (b) If successful, return *success*.
6. Undo all implications.
7. Place the line back on the j-frontier.
8. Set and first-level-justify the chosen input to $\bar{c}$.
9. If successful
   (a) Call *d_alg()*.
   (b) If successful, return *success*.
10. Undo all implications and return *fail*.

```
int just_act_xor(imply_list,line,val)
imply_rec *imply_list ;
int line , val ;
```

This function justifies the output of an *exclusive-OR* gate. Since we allow only two-input *XOR* gates in our circuits, if both inputs are free we have two options with which to justify the output. If only one input is free, we have just one way in which to justify the output. The function steps are

1. Take the line off the j-frontier.
2. If both inputs are free
   (a) Choose an assignment pair.
   (b) First-level-justify the pair.
   (c) If successful, call *d_alg()*.
   (d) If successful, return *success*.
   (e) Undo all implications.
   (f) Change the assignment pair and first-level-justify.
   (g) If successful, call *d_alg()*.
   (h) If successful, return *success*.
   (i) Undo all implications.
   (j) Add the line to the j-frontier and return *fail*.
3. If one input is free
   (a) Assign the free input the required value and first-level-justify.
   (b) If successful, call *d_alg()*.
   (c) If successful, return *success*.
   (d) Undo all implications.
   (e) Add the line to the j-frontier and return *fail*.

```
int just_and(gate)
int gate ;
```

The function receives the name of an *AND* gate and returns the name of the line chosen to be set to the controlling value of the gate. The selection process is done by going over the input lines as they appear in the fan table and selecting the first input line with an x value.

```
int just_nand(gate)
int gate ;
```

Same as above, for a *NAND* gate.

```
int just_nor(gate)
int gate ;
```

Same as above, for a *NOR* gate.

```
int just_or(gate)
int gate ;
```

Same as above, for an *OR* gate.

```
int just_xor(gate,val,t)
int gate , val , t ;
```

When trying to justify the output of a two-input *XOR* gate with two free inputs, for each output value we have two possible input assignments. Our strategy is to first try $(0, 0)$ and then try $(1, 1)$ when the output is 0 and to first try $(0, 1)$ and then $(1, 0)$ when the output is 1. Depending on the value of t (for *try*), we try to first-level-justify the $t-$th option. If successful we return *success*, otherwise we return *fail*. The function steps are

1. If $t == 1$
   (a) If $val == 0$
      i. *imply_and_check()* $(0, 0)$ as the input pair.
      ii. If successful return *success*, else return *fail*.
   (b) $val == 1$
      i. *imply_and_check()* $(0, 1)$ as the input pair.
      ii. If successful return *success*, else return *fail*.
2. $t == 2$
   (a) If $val == 0$

      i. *imply_and_check()* $(1,1)$ as the input pair.
      ii. If successful return *success*, else return *fail*.
   (b) $val == 1$
      i. *imply_and_check()* $(1,0)$ as the input pair.
      ii. If successful return *success*, else return *fail*.

```
remove_from_d_front(gate)
int gate ;
```

If *gate* is a primary output, *d_out_count* is decremented. Otherwise, *d_front_count* is decremented and the d-frontier list is scanned from the end backwards, until *gate* is found. It is then removed from the list.

```
remove_from_j_front(line)
int line ;
```

The function receives the name of a line and removes it from the j-frontier. It follows the list from the last element towards the first until it finds the given line.

```
undo(imply_list)
imply_rec *imply_list ;
```

The purpose of this function is to undo all implication due to a cancelled decision. This include setting all implied line values back to x, taking all the implied line justifications off the j-frontier and restoring the state of the d-frontier. At any one time, because there are many levels of recursion and every level has its own implication list, there exists several such lists. When adding implications to the list, we use a global variable that points to the end of the list of the current level. When backtracking is done, the *current* level changes, and the global pointer no longer points to the end of the *new* current list. Therefore we first assign the global pointer the value of the pointer to the first element, which is a local variable of the level. We then traverse the list to the last element, so that the *undoing* is in the reverse order of the *doing*. Every element in the list has a name of a line and the action performed on the line. If the action is *imply*, we reset the line value to x and update the *logic_count[]* of the destination gate. If the assignment was a fault value, we add the source gate to the d-frontier. If the destination gate has no other faulty inputs, we take it off the d-frontier. If the action is *justify*, we take the line off the j-frontier. The function steps are

1. Set a pointer to the end of the implication list.
2. While the pointer points to a structure
   (a) If the action is *justify*, remove the line from the j-frontier.
   (b) Otherwise (the action is *imply*)

i. Change the line value to x and update the *logic_count[]* of the destination gate.

ii. If the value was d or *dbar*

A. Add the source gate to the d-frontier.

B. If there is no other faulty input to the destination gate, take it off the d-frontier.

(c) Set the global pointer *imply_ptr* to point to *imply_list* and initialize this first structure.

Through most of the functions there are calls to special message routines. These messages print either error messages, output messages or, if the *trace* or *step* flags are *true*, a step by step account of all actions. For a description of these messages, see Appendix.

4.3.3 An example program

The example program is in the file *gen_tests.c* in the *d_alg* directory.

```
#include <stdio.h>

#include "d_alg.h"
#include "d_alg.e"
#include "/home/poisson3/ee658/global.g"

main(argc,argv)
int argc ;
char *argv[] ;
{ FILE *circuit , *fault_list ;
  int ok ;

  circuit = fopen(argv[1],"r") ;
  fault_list = fopen(argv[2],"r") ;
  if (argc == 3)
    { trace = false ;
      step = false ;
    }
  else
    { if (strcmp(argv[3],"step") == 0)
        { step = true ;
          trace = true ;
        }
      else
        if (strcmp(argv[3],"trace") == 0) trace = true ;
    }
```

```
  ok = parse(circuit) ;

  if (ok == success) gen_tests(fault_list) ;
}
```

The program is called with two or three parameters. The first is the circuit description file. The second contains a list of faults we want to generate tests for. The third is optional. When it is used, it is either the *trace* or *step* parameters, which cause the program to work in the selected mode. After openeing both files in read mode and checking the third parameter, the program parses the description file. If successful, it calls *gen_tests()* which generates tests for the faults in the fault list file.

5 Circuit simulation

The good circuit simulator is a 3-valued event-driven logic simulator. It is based on the 1-pass algorithm and uses a simple transport delay model for timing. It has the capability of simulating sequential circuits, provided that the *initial conditions* are specified properly.

5.1 Design of the simulator

While implementing an event-driven good circuit simulator, the following points must be considered:

1. Maintaining an event-list which specifies the events which take place at a specific instant of time. An event is basically a em line - value pair. With each instant of time, there is associated a list of such *line* - *value* pairs, indicating the values which those lines assume at that instant of time.

2. Maintaining a set of gates to be evaluated after every event. These gates correspond to the gates which are activated by the event being processed. This set is known as the *activated_set*. In the case of a 1-pass simulator, this set will always contain one element, since fanout points are treated as multiple-output buffers in this system.

3. Maintaining a set of events to be scheduled as a result of evaluating the gates in the *activated_set*. An event being processed causes other line values to change which constitute new events. This set, referred to as the *scheduled_ events_set*, consists of the new events and the time at which they are scheduled to take place.

4. In addition to the above, the simulator should have the capability of keeping a log of changes in line values with time, displaying values of certain signals, specifying initial conditions and specifying the length of simulation.

This particular implementation has functions which implement all of these capabilities.

1. Initialisation routines to initialise the various data structures and variables. The circuit has to start from some initial state. Unless specified by the user all lines are set to the unknown value x. These routines initialise the line values to the unknown value and initial values specified by the user.

2. Routines to manage and manipulate the event-list. The *event_list* changes dynamically during the simulation. The managing functions include initialising the event-list, picking up an event to be processed and adding new events to the list.

3. Routines to create and maintain the *activated_set* and the *scheduled_events_set*.

4. A routine to control the overall simulation and co-ordinate all the above routines. The controller can be modified for a 2-pass implementation. Features like hazard detection and oscillation control also can be performed by modifying the controller suitably.

The event-driven simulation algorithm is as follows:

$activated\_set = \Phi$
for every event $(i, v_i^{'})$ pending at current *time* t
    begin
    $v(i) = v_i^{'}$
    for gate j connected to line i
        begin
            update input values of j
            add j to $activated\_set$
        end
    for every $j \in activated\_set$
        begin
            $v_j^{'} =$ evaluate(j)
            if $v_j^{'} = lsv(j)$ **then**
              begin
              schedule $(j, v_j^{'})$ for time $t + d(j)$
              $lsv(j) = v_j^{'}$
              end
        end
    end

Before going into a detailed description of the algorithms and the routines, a brief overview of the simulator is presented here.

The simulator has four main modules:

1. The event-list manager

2. The event processor
3. The evaluator
4. The scheduler

For every event, the simulator loops through these four modules. The function of each module is as follows:

1. The event-list manager: This module performs functions to manipulate the event-list, depending upon the task it is asked to do. As mentioned earlier, the *event_list* is an array of pointers to a linked list of events. Each node on the list is a structure called *event* which has two fields - *line* and *value*. The following three tasks are performed by the:
   (a) It initialises the event-list. The user specifies a forcing function at the inputs, i.e. behavior of inputs with respect to time. This can be looked at as a list of events which take place at the inputs and are already known before the simulation begins. During initialisation the input events are added to the event-list.
   (b) It also picks (deletes) an event from the list for processing. Two global pointers, *present_event* and *next_event* point to the current and the next event in the *event_list*. After the *present_event* is completely processed, it's value is set to that of *next_event*. If this value is *null*, then the *time* is advanced and the first event for that *time* becomes the *present_event*.
   (c) It adds newly scheduled events to the list. In the 1-pass simulation algorithm, for every event, the *scheduled_events_set* is created which represents the new events resulting from processing the *present_event*. One of the subtasks of this task is to ensure that only *true events* are scheduled or added to the *event_list*. Before adding an event, the manager checks to see the last scheduled value, $lsv(line)$, and if the new value is the same, the event is not added to the *event_list*.
2. The event processor: The global pointer *present_event* always points to the current event. This module processes the current event. Processing the event involves three distinct tasks:
   (a) Set the *value* field of the *line* corresponding to the event.
   (b) Log the event in the *history* of the line as a *time* - *value* pair.
   (c) Change the input logic-value count of the *gate* of which the *line* is an input.
   (d) Find the activated gate and add it to the *activated_set*.
3. The evaluator: This module uses the *activated_set* as the input and evaluates all the gates in the set. It computes the output values of the gates and creates a set of new events - the *scheduled_events_set*. The evaluation method used in this simulator is based on input counting.

4. The scheduler: The scheduler schedules the events in the *scheduled_events_set* to occur at particular time. The time of the event is calculated using the delay model and the set of new events is passed on to the event-list manager so that the events can be added to the *event_list*.

On top of these modules is the controller module which controls the execution of these modules. The controller is also responsible for advancing the simulation time. If the controller configures the modules to run sequentially for every event, we have a 1-pass simulator. But if the controller configures them in such a way that only the *event-list manager* and the *event_processor* loop for all the events during a particular timestep and then the *evaluator* and *scheduler* loop till the *activated_set* is empty, we get a 2-pass simulator.

The controller runs the simulation for a period of time specified by the user.

5.2 Circuit simulator functions

5.2.1 Event-list manager

```
add input_event(time, line, value)
int time, line, value;
```

This function modifies the global data structure - the event-list. It adds the event specified by the *line* and *value* to the event-list corresponding to the timestep specified by *time*.

```
record_ip_events(input_gate)
int input_gate;
```

This function is used to initialise the event-list. The input events at the *input_gate* are added to the event-list by this function.

```
initialize_event_list()
```

This function reads the input events (waveforms) specified by the user in the control file and initialises the event-list.

```
add_events()
```

This function adds the newly scheduled events (i.e. elements of the scheduled-events-set) to the event-list.

```
pick_events()
```

This function deletes the current event from the event-list.

```
event_list_mngr(task)
int task;
```

This function performs a particular task as specified by *task*. *task* could be *init, add* or *pick*.

5.2.2 Event-processor

```
update_value()
```

This function updates the value of the line corresponding to the *present_event*. It changes the *value* field of the associated line and the *logic_count* of the associated gate.

```
add_act_set(act_gate)
int act_gate;
```

This function adds the *gate* to the *activated_set*. A node of structure type *act_gate* is added to the set which is a singly linked list.

```
event_processor()
```

This function updates the state of the circuit according to the *present_event* and creates the *activated_set*.

5.2.3 Evaluator

```
add_sched_event (line, value)
int line, value;
```

This function adds the new event specified by *line, value* to the *scheduled_events_set*. A node of structure type *sched_event* is added to the set which is a singly linked list with the header *scheduled_events_set*.

```
evaluator()
```

This function evaluates every gate (computes the output value of every gate) in the *activated_set* and creates the *scheduled_events_set*.

5.2.4 Scheduler

```
scheduler()
```

This function calculates the time at which the new events are to be scheduled, using the delay model.

5.2.5 Other functions

```
simulate(pi_events)
FILE *pi_events;
```

This function controls the event-driven simulation. The body of the function consists of a loop which increments the simulation time until the duration of simulation. A pointer to the *control file* is passed to the function. The *control file* specifies the input events and the simulation duration. The function steps are:

For every event (*line, value*) at current time:

1. Pick the event from the event-list
2. Process the event - i.e. set the value of the line
3. Evaluate the gate fed by the line
4. If evaluation results in *true event*, schedule it.
5. Add newly scheduled events to the *event_list*.

```
int transport_delay(line, value)
int line, value;
```

This function calculates the delay associated with *line* to take on the *value*. The delay is associted with the gate of which the *line* is an output. The function returns the actual time at which the event is to be scheduled.

```
initialize(pi_events)
FILE *pi_events;
```

This function initialises the circuit before the simulation begins. Besides the many book-keeping functions it performs, it sets all the initial line values to x, sets the appropriate logic counts of the gates and initialises other data structures.

In addition to the functions mentioned above, there are other functions which do the job of reading in data and printing out the results. The code of these functions is self-explanatory.

5.3 An example program

The following program is in the file *gsim.c* in the *gsim* directory.

```
#include <stdio.h>
#include "sim.h"
#include "sim.e"
#include "/home/poisson3/ee658/global.g"
```

```
main(argc,argv)
int argc ;
char *argv[] ;
{ FILE *circuit , *ip_event_list ;
  int ok;

  circuit = fopen(argv[1],"r") ;
  ip_event_list = fopen(argv[2],"r") ;
  if (argc == 3)
    { trace = false ;
      step = false ;
    }
  else
    { if (strcmp(argv[3],"step") == 0)
        { step = true ;
          trace = true ;
        }
      else
        if (strcmp(argv[3],"trace") == 0) trace = true ;
    }

  ok = parse(circuit) ;

  if (ok == success)
   simulate(ip_event_list) ;
}
```

The program is called with the circuit description file and the control description file as parameters. It opens the circuit description file in the read mode and calls *parse* tp parse the circuit. If parsing is successful, it calls the function *simulate(ip_event_list)* passing a pointer to the control description file as a parameter. This function simulates the circuit according to the control information specified in the control description file and prints out the results.

6 Fault simulation

The fault simulator is a 3-valued deductive simulator. In a deductive simulator, there is a fault list associated with every line. The basic step in deductive simulation is to compute the fault-free output value and the output fault-lists. The computation of fault lists is called fault-list propagation.

6.1 Design and implementation

The most important point to be considered in a deductive fault simulator is the representation of fault lists and fault-list propagation. Once the fault list representation is decided upon, routines for list manipulations can be easily written.

This particular implementation has the following features:

1. There is a fault list associated with every line. It is implemented as an array of *integers*, with each *bit* associated with a line. If a fault on any line is present in the fault list, the particular bit in the fault list is set to 1. Note that in a deductive simulator it is not necessary to store the actual fault value of the line in the fault list. It is enough to know the presence of a faulty line in a fault list. The actual fault can be determined by observing the good value of the particular line.

2. Macros are defined which perform bitwise manipulation of lists. FUNION, FINTERSECT, FCOMPLEMENT are the basic macros which implement union, intersection and complementation of lists.

3. A levelizer routine first assigns levels to all the gates and output fault lists are computed for gates at succeeding levels.

4. Routines are provided for generating various tests like *exhaustive, random* and *weighted random*. The system also provides a link between the test generator, described in an earlier chapter, and the fault simulator.

Before going on to a detailed description of the algorithms, a brief overview of the fault simulator is provided here.

There are three main tasks which the fault simulator performs

1. Initialisation

2. Test vector generation

3. Fault list propagation

These tasks are explained in detail below:

1. During initialisation the circuit is levelized. All gates are assigned a level, with primary inputs having level 0. The fault lists are initialised to have a fault corresponding to that line itself. The logic values of all the lines are set to x.

2. Depending upon the user choice, a set of test vectors is generated. The simulator writes these vectors to a file and then reads them one by one and simulates them. This test generation routines should not be confused with the D-algorithm based test generator which a part of **LTEST**. The D-algorithm based test generator comes up with a test for a particular fault. The test generation routines here create a set of *exhaustive*, *random* or *weighted random* for the purpose of fault simulation.

3. The core of the simulator is fault list propagation. For every test vector, a logic simulation is first performed. For every gate Z, the output fault list is computed as follows:
Let I be the set of inputs of gate Z with controlling value c and inversion i. Let C be the set of inputs with value c.

$$\textbf{if } C = \phi \textbf{ then } L_Z = \{\textstyle\bigcup_{j \in I} L_j\} \cup \{Zs\_a\_(c \oplus i\}$$
$$\textbf{else } L_Z = \{\textstyle\bigcap_{j \in C} L_j\} - \{\textstyle\bigcup_{j \in I-C} L_j\} \cup \{Zs\_a\_(\overline{c} \oplus i\}$$

That is if, no input has value c, any fault effect on an input propagates to the output. If some inputs have value c, only a fault that affects all the inputs at c without affecting any of the inputs at $\overline{c}$ propogates to the output. In both cases the local fault at the output is added.

6.2 Fault simulator functions

6.2.1 Initialisation routines

```
int compute_level(gate_index)
int gate_index;
```

This function returns the level of the gate specified by *gate_index*. The level of the gate is one more than the maximum of the levels of the gates feeding this gate.

```
levelize()
```

This function levelizes the circuit by calling *compute_level(gate_index)* recursively. The level field of every gate is assigned an integer value.

```
ch_dest_gate_lcount(line_ind, gate_ind)
int line_ind, gate_ind;
```

This function changes the logic count of the gate *gate_ind* as a result of one of it's input lines, *line_ind* changing it's value.

```
initialize_ckt()
```

This function initialises the circuit. It computes initial logic values, logic counts and does some other book-keeping operations.

```
change_d_db()
```

This function changes any line values which are *d* to 1 and *dbar* to 0. This is necessary when the fault simulator is used in conjunction with the test generator, since the fault simulator recognises only 3 logic values.

6.2.2 Test vector generation routines

```
gen_exhaust_test()
```

This function creates all possible input vectors and writes them to a file named *exhaust*.

```
char* rand_vector(seed,sig_pr)
int *seed;
float sig_pr[];
```

This function returns a pointer to a string which is a random vector. The routine uses the seed pointed to by *seed* and the input signal probabilities specified by the array *sig_pr*.

```
gen_random_test()
```

This function cretes as many random vectors as specified by the user. It calls function *rand_vector(seed,sig_pr)* for every vector. All the elements in the array *sig_pr* are initialised to 0.5 for truly random vectors. The vectors are writen to a file called *rand*.

```
gen_wtd_test()
```

This function creates weighted random vectors. The input signal probabilities are assigned by the user. The vectors are written to a file named *wtd_rand*.

6.2.3 Fault propagation routines

```
activate_faults()
```

This function initialises the fault lists. For every line, the bit in the fault list corresponding to the line is set.

```
get_f_list(gate, f_list_c, f_list_nc)
int gate;
int f_list_c[MAXINTS], f_list_nc[MAXINTS];
```

This function computes the controlling and the non-controlling fault lists for *gate*. The steps in this function are:

1. Depending upon the type of *gate*, set controlling value, *c_val*.
2. For every input i of *gate*

(a) If input value, *in_val* $= X$, do nothing.

(b) If *in_val* $=$ *c_val* then compute $L_i \cap f\_list\_c$.

(c) If *in_val* $\neq$ *c_val* then compute $L_i \cup f\_list\_nc$.

What this function effectively does is compute $\{\bigcup_{j \in I} L_j\}, \{\bigcap_{j \in C} L_j\}$ and $\{\bigcup_{j \in I-C} L_j\}$, where I is set of inputs and C is set of inputs with controlling value.

```
fpropogate(gate)
int gate;
```

This function computes the output fault list of *gate*. It calls routine *get_f_list(gate, f_list_c, f_list_nc)* to compute $\{\bigcup_{j \in I} L_j\}, \{\bigcap_{j \in C} L_j\}$ and $\{\bigcup_{j \in I-C} L_j\}$. The steps are:

1. If *gate* is of type *inv*, $L_Z = L_{input} \bigcup L_Z$.
2. If *gate* is of type *buf*, for every output j, $L_{Z_j} = L_{input} \bigcup L_{Z_j}$.
3. If *gate* is of any other type:
   if $C = \phi$ **then** $L_Z = \{\bigcup_{j \in I} L_j\} \bigcup \{L_Z\}$
   else $L_Z = \{\bigcap_{j \in C} L_j\} - \{\bigcup_{j \in I-C} L_j\} \bigcup \{L_Z\}$

```
fault_sim()
```

This function performs output fault list propagation for every gate in the circuit. The fault list corresponding to the primary output gates, is the list of detected faults. It also prints the fault lists of the primary outputs which are the faults detected.

6.2.4 Other functions

```
drop_fault(linenum, sa_value)
int linenum, sa_value;
```

This function drops the fault, *line* stuck-at *sa_value* from all the fault lists. So the next test vector does not simulate these faults again.

In addition to the functions mentioned above, there are other functions which do the job of reading in data and printing out the results. The code of these functions is self-explanatory.

6.3 An example program

The following program is in the file *fault_sim.c* in the *fsim* directory.

```
void fault_sim()
{
  int count, gate_index, level;


  change_d_db();
  activate_faults();
  ckt_depth = levelize();

  for (level = 1; level < ckt_depth; level++)
    for (gate_index = 0; gate_index < first_pi; gate_index++)
      if (gate_list[gate_index].level == level)
        fpropogate(gate_index);
  print_det_faults();
}
```

The function description is provided in the previous section.

7 User manual

7.1 Input format

7.1.1 Circuit description file

A circuit description file is made up of four parts, one of which is optional. The ordering of these parts is fixed and can not be switched. At the end of the file the keyword *end* must be specified. Throughout the file, blanks, empty lines and *C*- like comments (over one or several lines) can be added anywhere. Keywords can be written in either lower or upper case. No keyword can be used as a name for either a gate or a line. When specifying an input line format, optional parameters will be placed in square brackets ([]). The order in which the input line parameters are presented is fixed and can not be changed.

The first part of the input file is the *clock declaration segment*. It begins with a *declaration line* with the keyword *clocks*. The declaration line is followed by lines with clock names, one name per line. This part is optional, since not all circuits use clocked elements.

The second, third and fourth segments must appear in every input file.

The second segment is the *gate declaration segment*. It begins with a *declaration line* with the keyword *gate*. In each of the following lines a gate (combinational or clocked) is declared, one gate per line. All user names containing more than seven characters will be truncated to include only their seven character prefix. The format of a line varies, depending on the type of the declared gate.

If the gate is of type *AND, OR, NAND, NOR, INV* or *XOR*, the format is

gate_name gate_type output_line input_lines [rise_delay] [fall_delay] *gate_name* is the user specified name for the declared gate.

gate_type is the keyword (*and, or, nand, nor, inv, xor*) representing the type of the declared gate.

output_line is the user name for the line connected to the output of the declared name.

input_lines is a list of the user names for the lines connected to the inputs of the declared gate.

The *rise_delay* (*fall_delay*) field is optional and is used only when the *rise* (*fall*) delay of the gate is different than the default value for the gate type. A delay parameter is specified by the keyword *rise* (*fall*) followed by the delay value. If both parameters are specified, the *rise* parameter must come before the *fall* parameter.

If the gate is an *inverter*, the fan-in is one. If the gate is an *exclusive-OR*, we allow only two input lines. All other gates can have up to *max_fan* fan-in, where the default value for *max_fan* is six. If this parameter is changed, the whole system must be recompiled.

To represent branch points a gate of type *BUF* is used. The input format is

gate_name BUF input_line output_lines [rise_delay] [fall_delay]

The fields are the same as above, the difference begin that since there is only one input line and multiple output lines, the name of the lone input line comes before the names of the output lines. The limit on the number of output lines is *max_fan*.

When the gate type is *DFF*, the input format is

gate_name DFF clock_line [reset_line] [preset_line] d_line [q_line] [qbar_line] [rise_delay] [fall_delay] [setup_time] [hold_time]

gate_name is the user name for the gate.

clock_line is the name of the clock driving the gate. It must be a name of a clock that was previously declared in the *clock declaration segment*.

The *reset_line* (*preset_line*) field is optional and used to represent the name of the line connected to the *reset* (*preset*) input of the gate. The field is specified by the keyword *rl* (*pl*) followed by the line name.

d_line is the name of the line connected to the *d* input of the gate.

The *q_line* and *qbar_line* fields are optional and represent the name of the line connected to the *q* or *qbar* output of the gate. They are optional since one of these outputs might not be used, but since one of them is always used, at least on of these fields must be present. The field is specified by the keyword *q* (*qb*) followed by the line name.

The delay parameters *rise* and *fall* are the same as for combinational gates, but for clocked elements we have two additional parameters, the *setup* and *hold* requirements.

When the gate type is *JKFF*, the format is

gate_name JKFF clock_line [reset_line] [preset_line] j_line k_line [q_line] [qbar_line] [rise_delay] [fall_delay] [setup_time] [hold_time]

All fields are the same as for a *D flip flop*, the difference being in the type specified and that in place of a *d* line we specify the *j* and *k* lines.

The third segment is the *primary input segment*. It begins with a *declaration line* with the keywords *primary inputs*. This line is then followed by lines of the format

primary_input_gate primary_input_line

A primary input is modelled as a one output gate. The *primary_input_gate* is the name of the primary input and the *primary_input_line* is the name of the line connected to the primary input.

The fourth and final segment is the *primary output segment*. It begins with a *declaration line* with the keywords *primary outputs*. This line is then followed by lines of the format

primary_output_gate primary_output_line

A primary output is modelled as a one input gate. The *primary_output_gate* is the name of the primary output and the *primary_output_line* is the name of the line connected to the primary output.

This section is followed by the keyword *end* which specifies the end of the circuit description file.

An example circuit description:

```
/* fig 6.7 p. 186 */
gates
b1 buf a a1 a2
b2 buf c c1 c2
b3 buf i i1 i2
g1 nand i a2 b c1
g2 nor h d a1
g3 inv j c2
g4 nor k j e
g5 and f1 h i1
g6 and f2 i2 k
primary inputs
a a
b b
c c
d d
e e
primary outputs
f1 f1
f2 f2
end
```

7.1.2 Control description file

The control description file is used only by the good circuit simulator. It specifies the control information. It consists of three parts, the first of which is essential. The other two are optional.

The first part consists of the input waveforms specified as a list of events. For each input the input *line name* is specified followed by the sequence of events within round brackets. Each event is specified as a time-value pair. Events are delimited by a ":". The input event list is followed by an integer which specifies the duration of simulation.

The first optional part consists of *initial conditions* followed by the reserved word *ic*. *Initial conditions* are values that the user can force some lines to have. They also represent the initial states in the case of sequential circuits. Each line is a *line name* followed by the initial value.

The second optional part consists of a list of line names whose values are to be printed in the *trace* mode. This can be specified by a line which begins with the reserve word *print* and is followed by *line names*.

The control description file must have the reserved word *end* at the end of the control description.

An example control description file:

```
s (0 1: 5 2: 10 0)
r (0 1: 5 2: 10 0)
20
ic
y 0
y1 0
y2 0
ybar 1
ybar1 1
ybar2 1
print y ybar
end
```

7.1.3 Fault list file

The fault list file is used by the test generator as well as the fault simulator.

For the test generator it specifies the faults for which tests are desired and for the fault simulator it specifies the faults to be simulated.

The file is a list of faults, one fault per line. Each line is thus the *line name* followed by the *stuck at value*.

An example fault list file:

```
a 0
a 1
b 1
c 0
```

```
c 1
d 1
e 0
g 0
h 0
h 1
```

7.2 System commands

This section lists the commands the system recognizes. For each command the command line format and the resulting output is specified.

`collapse filename [> output_filename]`

The command receives the name of a circuit description file as a parameter. It parses the circuit file and finds all non-dominating and dominating equivalent fault classes. The output is the equivalence table that was constructed and a list of all non-dominating classes followed by a list of all dominating classes. For each class we specify the class representative and if there is a dominating fault we specify that fault. The output is sent to the standard output device but can be redirected to a user specified output file.

`gen_tests circuit_file fault_list [step] [trace] [> output_filename]`

The command receives the name of a circuit description file and a name of a file containing a list of faults, one fault per line (the fault list format is *line_name stuck_at_value*, where *line_name* is the user name for the faulty line and *stuck_at_value* is either 0 or 1). If neither of the *step* or *trace* parameters are specified, the output is a list of test vectors for non-redundant faults, and if a fault is redundant, a message stating this fact is generated. If the *step* parameter is used, all actions are single stepped, informing the user of every action that is taking place. Before proceeding with any selected option, the selected option is presented, along with a list of all other options, and the user can either continue processing with the selected option, or change it. In the *trace* mode, all action are stated but there is no user interaction. Only one of these parameters can be used.

`min_set filename [> output_file_name]`

The command receives the name of a circuit description file. It's function is to generate a minimal test set for all non-redundant stuck at faults in the circuit. The generated set is minimal in the sense that if we discard any vec- tor in the set, some non-redundant fault will not be caught. The function parses the circuit and does fault collapsing. It then iterates over the fault list until this list is emptied. For each fault taken off the list the function first tries to generate a test vector. If one is found, fault simulation is also performed and all other caught faults that are on the fault list are taken off. If a test vector can not be generated, the fault is added to the redundant fault list. At the end of the test set generation, all redundant faults are listed.

`parse filename [> output_filename]`

The command receives the name of a circuit description file. It parses the file and creates the global data structures. The output is a printout of the global structures *gate_list[], line_list[], fan_table[][], line_x_reference[]* and *clk_table[]*. It also outputs the initial list of fault classes representatives.

`test_set filename [> output_filename]`

The command receives the name of a circuit description file. It parses the file, performs fault collapsing and generates tests for all non-redundant faults. The output is a list of test vectors for all non-redundant faults and a list of all redundant faults.

`gsim circuit_file control_file [step] [trace [> output_filename] ]`

The command receives the name of a circuit description file and the name of a file containing the control information for the simulation. The control_file should contain the primary input events, simulation duration and line values to be printed out in the *trace* mode. If neither the *step* or *trace* parameters are used only each event being simulated is printed out. If the *step* option is used, all actions are single stepped - the simulator halts after every event. Depending upon user selected options, information is displayed after every event. If the *trace* parameter is used, the simulator runs through the simulation, stating events processed and the final results. There is no user interaction in this mode. The redirection option can be used only with the *trace* parameter.

```
fsim circuit_file [fault_list_file] [step]
     [ trace [type_of_test [number_of_vectors]] [> output_filename] ]
```

The command receives the name of a circuit description file. The second argument is the name of a file with the faults to be simulated (the fault list format is *line_name stuck_at_value*, where *line_name* is the user name for the faulty line and *stuck_at_value* is either 0 or 1). If this argument is not specified, all stuck-at faults are simulated. If the *step* parameter is used, the simulator steps through every test vector simulated. The user can check any fault list and see the faults detected. The *step* mode also asks the user for the type of test. If the *trace* parameter is used, the *type_of_test* argument has to follow. This argument could be a file name consisting of the test vectors or the reserved words *exhaust*, *rand* or *wtd*. If the *type_of_test* is specified as *rand* or *wtd* then the *number_of_vectors* parameter should be specified which has to be a 2-digit number. (For e.g. if 4 random vectors are needed, specify 04 as the parameter).

Both *step* and *trace* cannot be used together and one of them has to be specified at all times.

A Input grammar

```
circuit : hdr clocks gates primary_inputs primary_outputs
        ;

hdr : nl
    |
    ;

nl : NL
   | nl NL
   ;

clocks : CLOCKS nl clock_list
       |
       ;

clock_list : clock
           | clock_list clock
           ;

clock : NAME nl
      ;

gates : GATES nl gate_list
      ;

gate_list : gate
          | gate_list gate
          ;

gate : gate_name gate_type nl
     ;

gate_name : NAME
          ;

gate_type : DFF ff_lines d q qbar delay
          | JKFF ff_lines j k q qbar delay
          | AND oi_list delay
          | OR oi_list delay
          | INV oi_list delay
          | NAND oi_list delay
          | NOR oi_list delay
```

```
          | XOR oi_list delay
          | BUF io_list delay
          ;

ff_lines : clock_line preset_line reset_line
         ;

clock_line : NAME
           ;

preset_line : PL NAME
            |
            ;

reset_line : RL NAME
           |
           ;

d : NAME
  ;

q : Q NAME
  |
  ;

qbar : QB NAME
     |
     ;

j : NAME
  ;

k : NAME
  ;

delay : rise fall setup hold
      ;

rise : RISE NUM
     |
     ;

fall : FALL NUM
     |
     ;
```

```
setup : SETUP NUM
      |
      ;

hold : HOLD NUM
     |
     ;

oi_list : output_line input_list
        ;

output_line : NAME
            ;

input_list : line_name
           | input_list line_name
           ;

line_name : NAME
          ;

io_list : input_line output_list
        ;

input_line : NAME
           ;

output_list : line_name
            | output_list line_name
            ;

primary_inputs : pi_head nl pi_list
               ;

pi_head : PRIMARY INPUTS
            }
        ;

pi_list : pi_name pi_line nl
        | pi_list pi_name pi_line nl
        ;

pi_name : NAME
        ;
```

```
pi_line : NAME
        ;

primary_outputs : po_head nl po_list end
                ;

po_head : PRIMARY OUTPUTS
        ;

po_list : po_name po_line nl
        | po_list po_name po_line nl
        ;

po_name : NAME
        ;

po_line : NAME
        ;

end : END nl
    ;
```

B Function-Function reference

B.1 Test generation

```
int add_line(user_name)
char *user_name ;
```

The function appears in the file *parser.y*.
The function calls

1. gen_err()
2. hash_name()

The function is called by

1. The *Yacc* output.

The function reads the global variables

1. line_x_reference[]

The function assigns values to the global variables

1. line_list[]
2. line_x_reference[]
3. number_of_lines

```
add_to_d_front(gate)
int gate ;
```

The function appears in the file *lists.c*.
The function calls no other functions.
The function is called by

1. drive_act()
2. imply_and_check()
3. undo()

The function reads the global variables

1. d_front_last
2. gate_list[]

The function assigns values to the global variables

1. d_front_last
2. d_front_count
3. d_out_count

```
add_to_dom_list(fault,line)
int fault , line ;
```

The function appears in the file *fc_lists.c*.
The function calls no other functions.
The function is called by

1. fc()

The function reads the global variables

1. dominant_list

The function assigns values to the global variables

1. dominant_list

```
add_to_eq_list(fault,line)
int fault , line ;
```

The function appears in the file *fc_lists.c*.
The function calls no other function.
The function is called by

1. fc()

The function reads the global variables

1. equivalence_list

The function assigns values to the global variables

1. equivalence_list

```
add_to_eq_reps(fault,line)
int fault , line ;
```

The function appears in the file *fc_lists.c*.
The function calls no other function.
The function is called by

1. fc()
2. The *Yacc* output.

The function reads the global variables

1. equivalence_reps

The function assigns values to the global variables

1. equivalence_reps

```
add_to_f_list(fault,line)
int fault , line ;
```

The function appears in the file *fc_lists.c*.
The function calls no other functions.
The function is called by

1. fc()
2. test_set()

The function reads the global variables

1. fault_list

The function assigns values to the global variables

1. fault_list

```
add_to_gate_list((gate,type)
int gate , type ;
```

The function appears in the file *parser.y*.
The function calls

1. gen_err()
2. gen_warn()

The function is called by

1. The *Yacc* output.

The function reads the global variables

1. gate_list[]
2. line_list[]

The function assigns values to the global variables

1. fan_table[][]
2. gate_list[]
3. line_list[]

```
add_to_imply_list(line,val,action)
int line , va , action ;
```

The function appears in the file *lists.c*.
The function calls

1. add_to_j_front()
2. do_message()

The function is called by

1. imply_and_check()

The function reads the global variables

1. imply_ptr

The function assigns values to the global variables

1. gate_list[]
2. imply_ptr
3. line_list[]

```
add_to_j_front(site)
int site ;
```

The function appears in the file *lists.c*.
The function calls no other functions.
The function is called by

1. add_to_imply_list()
2. just_act_gate()
3. just_act_xor()

The function reads the global variables

1. j_front_last

The function assigns values to the global variables

1. j_front_count
2. j_front_last

```
add_to_red_list()
```

The function appears in the file *test_set.c*.
The function calls no other functions.
The function is called by

1. test_set()

The function reads the global variables

1. redundant_list

The function assigns values to the global variables

1. redundant_list

```
imply_rec  choose_next_action()
```

The function appears in the file *choose.c*.
The function calls no other functions.
The function is called by

1. d_alg()

The function reads the global variables

1. d_front_last
2. d_out_count
3. j_front_last

The function assigns values to no global variables.

```
int compute_val(gate)
int gate ;
```

The function appears in the file *compute_val.c*.
The function calls no other functions.
The function is called by

1. imply_and_check()

The function reads the global variables

1. fan_table[][]
2. gate_list[]

The function assigns values to no global variables.

```
d_alg()
```

The function appears in the file *d_alg.c*.
The function calls

1. choose_next_action()
2. do_message()
3. drive_act()
4. just_act()
5. print_fronts()

The function is called by

1. drive_fault()
2. gen_test()
3. just_act()
4. just_act_gate()
5. just_act_xor()

The function reads the global variables

1. d_front_count
2. d_out_count
3. j_front_count

The function assigns values to the global variables

1. imply_ptr

```
dangling_lines()
```

The function appears in the file *parser.y*.
The function calls

1. gen_err()

The function is called by

1. parse().

The function reads the global variables

1. line_list[]
2. number_of_lines

The function assigns values to no global variables.

```
int drive_act(imply_list,gate)
imply_rec *imply_list ;
int gate ;
```

The function appears in the file *drive_act.c*.
The function calls

1. add_to_d_front()
2. drive_fault()
3. remove_from_d_front()
4. undo()
5. undo_message()

The function is called by

1. d_alg()

The function reads the global variables

1. gate_list[]

The function assigns values to the global variables

1. gate_list[]

```
int drive_fault(gate,dval,c,i)
int gate , dval , c , i ;
```

The function appears in the file *drive_act.c*.
The function calls

1. d_alg()
2. imply_and_check()
3. remove_from_d_front()

The function is called by

1. drive_act()

The function reads the global variables

1. fan_table[][]
2. gate_list[]
3. line_list[]

The function assigns values to no global variables.

```
do_message(num, site,val)
int num , site , val ;
```

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. add_to_imply_list()
2. drive_act()
3. d_alg()
4. just_act()
5. just_act_gate()
6. just_act_xor()

The function reads the global variables

1. gate_list[]
2. line_list[]
3. trace

The function assigns values to no global variables.

```
fail_message(num,line)
int num , line ;
```

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. imply_and_check()

The function reads the global variables

1. line_list[]
2. trace

The function assigns values to no global variables.

```
fc()
```

The function appears in the file *fc.c*.
The function calls

1. add_to_dom_list()
2. add_to_eq_list()
3. add_to_eq_reps()
4. add_to_f_list()
5. fc_warn()

The function is called by

1. main() (in *collapse.c*)
2. main() (in *gen_test_set.c*)

The function reads the global variables

1. active
2. dominant
3. equivalence_hdr
4. equivalence_list
5. equivalence_reps
6. fan_table[][]
7. first
8. gate_list[]
9. reps_hdr

The function assigns values to the global variables

1. active
2. dominant
3. equivalence_list
4. equivalence_reps
5. equivalence_table[][]
6. first

```
fc_warn(num,site)
int num , site ;
```

The function appears in the file *fc.c*.
The function calls no other functions.
The function is called by

1. fc()

The function reads the global variables

1. gate_list[]

The function assigns values to no global variables.

```
int find_gate_name(s)
char *s ;
```

The function appears in the file *parser.y*.
The function calls

1. gen_err()

The function is called by

1. The *Yacc* output.

The function reads the global variables

1. number_of_gates

The function assigns values to the global variables

1. gate_list[]
2. number_of_gates

```
gen_err(err_num,site_name,line_num)
int err_num ;
char *site_name ;
int line_num ;
```

The function appears in the file *parser.y*.
The function calls no other function.
The function is called by

1. add_line()
2. add_to_gate_list()
3. find_gate_name()
4. get_line()
5. The *Yacc* output.

The function reads no global variables.
The function assigns values to no global variables.

```
gen_lex_err()
```

The function appears in the file *lex.l*.
The function calls no other function.
The function is called by

1. lex.l, the *Lex* specification file.

The function reads no global variables.
The function assigns values to no global variables.

```
gen_lex_warn()
```

The function appears in the file *lex.l*.
The function calls no other function.
The function is called by

1. lex.l, the *Lex* specification file.

The function reads no global variables.
The function assigns values to no global variables.

```
int gen_test(site,flt)
```

The function appears in the file *test_gen.c*.
The function calls

1. d_alg()
2. imply_and_check()
3. init_search()

The function is called by

1. gen_tests()
2. test_set()

The function reads the global variables

1. line_list[]
2. number_of_gates

The function assigns values to the global variables

1. fault_gate
2. gate_list[]
3. imply_ptr
4. line_list[]

```
gen_tests(fault_list)
FILE *fault_list ;
```

The function appears in the file *test_gen.c*.
The function calls

1. gen_test()
2. gen_tests_init()
3. print_test()

The function is called by

1. main() (in *gen_tests.c*)

The function reads no global variables.
The function assigns values to no global variables.

```
gen_tests_init()
```

The function appears in the file *test_gen.c*.
The function calls no other functions.
The function is called by

1. gen_tests()
2. test_set()

The function reads the global variables

1. d_front_hdr
2. j_front_hdr

The function assigns values to the global variables

1. d_front_hdr
2. d_front_last
3. j_front_hdr
4. j_front_last

```
gen_warn(warn_num,site_name,line_num)
int warn_num ;
char *site_name ;
int line_num ;
```

The function appears in the file *parser.y*.
The function calls no other function.
The function is called by

1. add_to_gate_list()

The function reads no global variables.
The function assigns values to no global variables.

```
int get_gate(site)
int site ;
```

The function appears in the file *choose.c*.
The function calls no other functions.
The function is called by

1. show_step()

The function reads the global variables

1. gate_list[]
2. number_of_gates

The function assigns values to no global variables.

```
int get_line(user_name)
char *user_name ;
```

The function appears in the file *parser.y*.
The function calls

1. gen_err()
2. hash_name()

The function is called by

1. show_step()

The function reads the global variables

1. line_list[]
2. line_x_reference[]

The function assigns values to no global variables.

```
int hash_name(user_name)
```

The function appears in the file *parser.y*.
The function calls no other functions.
The function is called by

1. add_line()
2. get_line()

The function reads no global variables.
The function assigns values to no global variables.

```
imply_and_check(line,val,action)
int line , val , action ;
```

The function appears in the file *imply_and_check.c*.
The function calls

1. add_to_d_front()
2. add_to_imply_list()
3. compute_val()
4. fail_message()
5. imply_and_check()
6. imply_free_input()
7. remove_from_d_front()

The function is called by

1. drive_fault()
2. gen_test()
3. imply_and_check()
4. imply_free_input()
5. just_act_gate()
6. just_act_xor()
7. just_xor()

The function reads the global variables

1. fan_table[][]
2. fault_gate
3. gate_list[]
4. line_list[]

The function assigns values to the global variables

1. d_alg_error

```
imply_free_input(gate)
int gate ;
```

The function appears in the file *imply_and_check.c*.
The function calls

1. imply_and_check()

The function is called by

1. imply_and_check()

The function reads the global variables

1. fan_table[][]
2. gate_list[]

3. line_list[]

The function assigns values to the global variables

1. d_alg_error

`init()`

The function appears in the file *parser.y*.
The function calls no other function.
The function is called by

1. parse()

The function reads no global variables.
The function assigns values to the global variables

1. clk_cntr[]
2. clk_names[][]
3. line_list[]
4. line_x_reference[]

`init_fc()`

The function appears in the file *fc_list.c*.
The function calls no other functions.
The function is called by
The function reads the global variables

1. dominant_list
2. equivalence_list
3. equivalence_reps
4. fault_list

The function assigns values to the global variables

1. dominant_list
2. dominant_hdr
3. equivalence_hdr
4. equivalence_list
5. equivalence_reps
6. fault_hdr
7. fault_list
8. reps_hdr

`init_red_list()`

The function appears in the file *test_set.c*.
The function calls no other functions.
The function is called by

1. test_set()

The function reads the global variables

1. redundant_list

The function assigns values to the global variables

1. redundant_hdr
2. redundant_list

`init_search()`

The function appears in the file *test_gen.c*.
The function calls no other functions.
The function is called by

1. gen_test()

The function reads the global variables

1. d_front_hdr
2. fan_table[][]
3. first_pi
4. j_front_hdr
5. number_of_lines

The function assigns values to the global variables

1. d_alg_error
2. d_front_count
3. d_front_hdr
4. d_front_last
5. d_out_count
6. gate_list[]
7. j_front_count
8. j_front_hdr
9. j_front_last
10. line_list[]

```
int just_act(imply_list,line)
imply_rec *imply_list ;
int line ;
```

The function appears in the file *just_act.c*.
The function calls

1. d_alg()
2. do_message()
3. just_act_gate()
4. just_act_xor()
5. remove_from_j_front()

The function is called by

1. d_alg()

The function reads the global variables

1. gate_list[]
2. line_list[]

The function assigns values to no global variables.

```
int just_act_gate(imply_list,line,val)
```

The function appears in the file *just_act.c*.
The function calls

1. add_to_j_front()
2. d_alg()
3. do_message()
4. imply_and_check()
5. just_and()
6. just_nand()
7. just_nor()
8. just_or()
9. remove_from_j_list()
10. undo()
11. undo_message()

The function is called by

1. just_act()

The function reads the global variables

1. gate_list[]
2. line_list[]

The function assigns values to no global variables.

```
int just_act_xor(imply_list,line,val)
imply_rec *imply_list ;
int line , val ;
```

The function appears in the file *just_act.c*.
The function calls

1. add_to_j_front()
2. d_alg()
3. do_message()
4. imply_and_check()
5. just_xor()
6. remove_from_j_front()
7. undo()
8. undo_message()

The function is called by

1. just_act()

The function reads the global variables

1. fan_table[][]
2. line_list[]

The function assigns values to no global variables.

```
int just_and(gate)
int gate ;
```

The function appears in the file *justify.c*.
The function calls no other functions.
The function is called by

1. just_act_gate()

The function reads the global variables

1. fan_table[][]
2. line_list[]

The function assigns values to no global variables.

```
int just_nand(gate)
int gate ;
```

The function appears in the file *justify.c*.
The function calls no other functions.
The function is called by

1. just_act_gate()

The function reads the global variables

1. fan_table[][]
2. line_list[]

The function assigns values to no global variables.

```
int just_nor(gate)
int gate ;
```

The function appears in the file *justify.c*.
The function calls no other functions.
The function is called by

1. just_act_gate()

The function reads the global variables

1. fan_table[][]
2. line_list[]

The function assigns values to no global variables.

```
int just_or(gate)
int gate ;
```

The function appears in the file *justify.c*.
The function calls no other functions.
The function is called by

1. just_act_gate()

The function reads the global variables

1. fan_table[][]
2. line_list[]

The function assigns values to no global variables.

```
int just_xor(gate,val,t)
int gate , val , t ;
```

The function appears in the file *justify.c*.
The function calls

1. imply_and_check()

The function is called by

1. just_act_xor()

The function reads the global variables

1. fan_table[][]
2. line_list[]

The function assigns values to no global variables.

```
lex.l - the Lex specification file.
```

This is the file given to *Lex* from which he generates the lexical analyzer.
The function calls

1. gen_lex_err()
2. gen_lex_warn()

The functions created by *Lex*, from this file, are called by the parser generated by *Yacc*.
The function reads no global variables.
The function assigns values to no global variables.

```
main(argc,argv)   /* in collapse.c  */
int argc ;
char *argv[] ;
```

The function appears in the file *collapse.c*.
The function calls

1. fc()
2. parse()
3. print_eq_table()
4. print_fc()

The function is called by no other functions.
The function reads no global variables.
The function assigns values to no global variables.

```
main(argc,argv)   /* in gen_test_set.c  */
int argc ;
char *argv[] ;
```

The function appears in the file *gen_test_set.c*.
The function calls

1. fc()
2. parse()
3. test_set()

The function is called by no other functions.
The function reads no global variables.
The function assigns values to no global variables.

```
main(argc,argv)   /* in gen_tests.c  */
int argc ;
char *argv[] ;
```

The function appears in the file *gen_tests.c*.
The function calls

1. gen_tests()
2. parse()

The function is called by no other functions.
The function reads no global variables.
The function assigns values to the global variables

1. step
2. trace

```
main(argc,argv)   /* in parse.c  */
int argc ;
char *argv[] ;
```

The function appears in the file *parse.c*.
The function calls

1. parse()
2. print_tables()

The function is called by no other functions.
The function reads no global variables.
The function assigns values to no global variables.

```
int parse(circuit)
FILE *circuit ;
```

The function appears in the file *parser.y*.
The function calls

1. dangling_lines()
2. init()
3. yyparse()

The function is called by

1. main() (in *collapse.c*)
2. main() (in *gen_test_set.c*)
3. main() (in *gen_tests.c*)
4. main() (in *parse.c*)

The function reads no global variables.
The function assigns values to no global variables.

`parser.y - the Yacc specification file.`

This file is given to *Yacc*, from which it generates a parser.
The function calls made by the semantic routines are

1. add_line()
2. add_to_eq_reps()
3. add_to_gate_list()
4. find_gate_name()
5. gen_err()

The functions created by *Yacc* from this file are called by the generated parser.
The function reads the global variables

1. clk_cntr
2. gate_list[]
3. number_of_gates
4. number_of_lines

The function assigns values to the global variables

1. clk_cntr
2. clk_names[][]
3. clk_table
4. first_pi
5. first_po

`print_d_front()`

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. print_fronts()

The function reads the global variables

1. d_front_last
2. gate_list[]

The function assigns values to no global variables

`print_eq_tab()`

The function appears in the file *fc.c*.
The function calls no other functions.
The function is called by

1. main() (in *collapse.c*)

The function reads the global variables

1. equivalence_table[][]
2. line_list[]
3. number_of_lines

The function assigns values to no global variables

`print_fc()`

The function appears in the file *fc.c*.
The function calls no other functions.
The function is called by

1. main() (in *collapse.c*)

The function reads the global variables

1. dominant_hdr
2. dominant_list
3. equivalence_table[][]
4. fault_hdr
5. fault_list
6. line_list[]

The function assigns values to no global variables.

`print_fronts()`

The function appears in the file *print.c*.
The function calls

1. print_d_front()
2. print_j_front()

The function is called by

1. d_alg()

The function reads the global variables

1. trace

The function assigns values to no global variables.

`print_j_front()`

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. print_fronts()

The function reads the global variables

1. j_front_last
2. line_list[]

The function assigns values to no global variables.

`print_red_list()`

The function appears in the file *test_set.c*.
The function calls no other functions.
The function is called by

1. test_set()

The function reads the global variables

1. line_list
2. redundant_hdr
3. redundant_list

The function assigns values to no global variables.

`print_tables()`

The function appears in the file *parser.y*.
The function calls no other functions.
The function is called by

1. main() (in *parse.c*)

The function reads the global variables

1. clk_cntr[]
2. clk_names[][]
3. clk_table[][]
4. fan_table[][]
5. gate_list[]
6. line_list[]
7. number_of_gates
8. number_of_lines
9. reps_hdr

The function assigns values to no global variables.

```
print_test(ok,line_name,flt)
int ok ;
char *line_name ;
int flt ;
```

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. gen_tests()
2. test_set()

The function reads the global variables

1. first_pi
2. first_po
3. gate_list[]
4. line_list

The function assigns values to no global variables.

```
remove_from_d_front(gate)
int gate ;
```

The function appears in the file *lists.c*.
The function calls no other functions.
The function is called by

1. drive_act()
2. drive_fault()
3. imply_and_check()
4. undo()

The function reads the global variables

1. d_front_count
2. d_front_hdr
3. d_front_last
4. gate_list[]

The function assigns values to the global variables

1. d_front_hdr
2. d_front_count
3. d_out_count

```
remove_from_j_front(line)
int line ;
```

The function appears in the file *lists.c*.
The function calls no other functions.
The function is called by

1. just_act()
2. just_act_gate()
3. just_act_xor()
4. undo()

The function reads the global variables

1. j_front_hdr
2. j_front_last

The function assigns values to the global variables

1. j_front_count
2. j_front_hdr

```
show_step(action)
```

The function appears in the file *choose.c*.
The function calls

1. get_gate()
2. get_line()

The function is called by

1. d_alg()

The function reads the global variables

1. d_front_hdr

2. d_front_last
3. gate_list[]
4. j_front_hdr
5. j_front_last
6. line_list[]
7. step

The function assigns values to no global variables.

```
test_set()
```

The function appears in the file *test_set.c* .
The function calls

1. add_to_f_list()
2. add_to_red_list()
3. gen_test()
4. gen_tests_init()
5. init_red_list()
6. print_red_list()
7. print_test()

The function is called by

1. main() (in *gen_test_set.c*)

The function reads the global variables

1. equivalence_table[][]
2. fault_hdr
3. fault_list
4. line_list

The function assigns values to the global variables

1. equivalence_table[][]
2. fault_list

```
undo(imply_list)
imply_rec *imply_list ;
```

The function appears in the file *undo.c*.
The function calls

1. add_to_d_front()

2. remove_from_d_front()
3. remove_from_j_front()

The function is called by

1. drive_act()
2. just_act_gate()
3. just_act_xor()

The function reads the global variables

1. gate_list[]
2. line_list

The function assigns values to the global variables

1. gate_list[]
2. imply_ptr
3. line_list[]

```
undo_message(num,site,val)
int num , site , val ;
```

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. drive_act()
2. just_act_gate()
3. just_act_xor()

The function reads the global variables

1. gate_list[]
2. line_list[]
3. trace

The function assigns values to no global variables.

```
yyparse()
```

The function is a *Yacc* generated function.

B.2 Circuit simulation

```
void add_act_set(act_gate)
int act_gate;
```

The function appears in the file *event_processor.c*.
The function calls no other functions.
The function is called by

1. event_processor()

The function reads the global variables

1. activated_set

The function assigns values to global variables

1. activated_set

```
void add_events()
```

The function appears in the file *event_list_mngr.c*.
The function calls no other functions.
The function is called by

1. event_list_mngr()

The function reads the global variables

1. scheduled_event_set
2. event_list[]
3. time

The function assigns values to global variables

1. event_list[]
2. next_event
3. scheduled_event_set

```
void add_input_event(time, line, value)
int time, line, value;
```

The function appears in the file *event_list_mngr.c*.
The function calls no other functions.
The function is called by

1. record_ip_events()

The function reads the global variables

1. event_list[]

2. time

The function assigns values to global variables.

1. event_list[]

```
void add_sched_event (line, value)
int line, value;
```

The function appears in the file *evaluator.c*.
The function calls no other functions.
The function is called by

1. evaluator()

The function reads the global variables

1. line_list[]
2. scheduled_event_set
3. number_of_events
4. number_gates_act

The function assigns values to global variables.

1. scheduled_event_set

```
void ask_step()
```

The function appears in the file *print.c*.
The function calls

1. get_line()

The function is called by

1. simulate()

The function reads global variables

1. step
2. trace

The function assigns values to global variables

1. observe_lines[]
2. act_flag
3. event_flag
4. list_flag
5. list_flag

```
int compute_level(gate_index)
int gate_index;
```

The function appears in the file *init_ckt.c*.
The function calls

1. compute_level()

The function is called by

1. levelize()

The function reads global variables

1. gate_list[]
2. line_list[]
3. first_pi
4. first_po item number_of_lines

The function assigns values to global variables

1. gate_list[]

```
void evaluator()
```

The function appears in the file *evaluator.c*.
The function calls

1. compute_val()
2. add_sched_event()

The function is called by

1. simulate()

The function reads the global variables

1. gate_list[]
2. fan_table[][]
3. activated_set

The function assigns values to global variables.

1. activated_set

```
void event_list_mngr(task)
int task;
```

The function appears in the file *event_list_mngr.c*.
The function calls

1. initialize_event_list()

2. add_events()
3. pick_events()
4. show_step()

The function is called by

1. simulate()

The function reads no global variables
The function assigns values to no global variables

```
void event_processor()
```

The function appears in the file *event_processor.c*.
The function calls

1. update_value()
2. add_act_set()
3. shoe_step()

The function is called by

1. simulate()

The function reads the global variables

1. line_list[]
2. present_event
3. first_pi
4. number_gates_act

The function assigns values to global variables

1. present_event
2. number_gates_act

```
void initialize(pi_events)
FILE *pi_events;
```

The function appears in the file *init.c*.
The function calls

1. get_line()

The function is called by

1. simulate()

The function reads global variables

1. number_of_lines
2. number_of_gates

3. gate_list[]
4. line_list[]
5. fan_table[][]

The function assigns values to global variables

1. observe_lines[]
2. gate_list[]
3. line_list[]

`void initialize_ckt()`

The function appears in the file *init_ckt.c*.
The function calls

1. levelize()

The function is called by

1. simulate()

The function reads global variables

1. first_pi
2. first_po
3. logic_values
4. gate_list[]
5. line_list[]
6. number_of_gates

The function assigns values to global variables

1. line_list[]
2. gate_list[]
3. scheduled_event_set
4. activated_set

`void initialize_event_list()`

The function appears in the file *event_list_mngr.c*.
The function calls

1. record_ip_events()

The function is called by

1. event_list_mngr()

The function reads the global variables

1. event_list[]
2. first_pi
3. first_po

The function assigns values to global variables

1. event_list[]
2. number_of_events
3. number_gates_act

```
int levelize()
```

The function appears in the file *init_ckt.c*.
The function calls

1. compute_level()

The function is called by

1. initialize_ckt()

The function reads global variables

1. first_po item number_of_gates

The function assigns values to no global variables

```
main(argc,argv)
int argc ;
char *argv[] ;
```

The function appears in the file *gsim.c*.
The function calls

1. parse()
2. simulate()

The function is called by no other functions.
The function reads no global variables
The function assigns values to global variables

1. step
2. trace

```
void pick_events()
```

The function appears in the file *event_list_mngr.c*.
The function calls no other functions.
The function is called by

1. event_list_mngr()

The function reads the global variables

1. present_event
2. time
3. number_of_events

The function assigns values to global variables

1. event_list[]
2. number_of_events

`void print_act_set()`

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. show_step()

The function reads global variables

1. activated_set
2. gate_list[]

The function assigns values to no global variables.

`void print_curr\_event()`

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. show_step()

The function reads global variables

1. line_list[]
2. present_event

The function assigns values to no global variables.

`void print_event_list()`

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. show_step()

The function reads global variables

1. event_list[]
2. line_list[]
3. time
4. simulation_length

The function assigns values to no global variables.

`void print_lines()`

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. show_step()

The function reads global variables

1. line_list[]
2. observe_lines[]

The function assigns values to no global variables.

`void print_line_results()`

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. print_results()

The function reads global variables

1. line_list[]
2. observe_lines[]
3. simulation_length

The function assigns values to no global variables.

`void print_results()`

The function appears in the file *print.c*.
The function calls

1. print_line_results()

The function is called by

1. simulate()

The function reads global variables

1. line_list[]
2. gate_list[]
3. fan_table[][]
4. simulation_length
5. trace
6. first_pi
7. first_po
8. number_of_gates

The function assigns values to no global variables.

```
void print_sched_set()
```

The function appears in the file *print.c*.
The function calls no other functions.
The function is called by

1. show_step()

The function reads global variables

1. scheduled_event_set
2. line_list[]

The function assigns values to no global variables.

```
void read_input(ifp)
FILE *ifp;
```

The function appears in the file *read.c*.
The function calls no other functions.
The function is called by

1. simulate()

The function reads the global variables

1. first_pi
2. first_po
3. gate_list[]
4. line_list[]

The function assigns values to global variables

1. line_list[]
2. simulation_length

```
void record_ip_events(input_gate)
int input_gate;
```

The function appears in the file *event_list_mngr.c*.
The function calls

1. add_input_event()

The function is called by

1. initialize_event_list()

The function reads the global variables

1. gate_list[]
2. line_list[]

The function assigns values to no global variables.

```
void scheduler()
```

The function appears in the file *scheduler.c*.
The function calls

1. transport_delay()
2. show_step()

The function is called by

1. controller()

The function reads the global variables

1. scheduled_event_set

The function assigns values to no global variables

```
void show_step(num)
int num;
```

The function appears in the file *print.c*.
The function calls

1. print_act_set()
2. print_sched_set()
3. print_event_list()
4. print_lines()
5. print_curr_event()

The function is called by

1. simulate()

The function reads global variables

1. step
2. act_flag
3. event_flag
4. list_flag
5. line_flag
6. time

The function assigns values to no global variables.

```
void simulate(pi_events)
FILE *pi_events;
```

The function appears in the file *controller.c*.
The function calls

1. read_input()
2. initialize()
3. ask_step()
4. event_list_mngr()
5. event_processor()
6. evaluator()
7. scheduler()
8. show_step()
9. print_results()

The function is called by

1. main()

The function reads the global variables

1. event_list[]
2. time
3. next_event

The function assigns values to global variables.

1. present_event

```
int transport_delay(line, value)
int line, value;
```

The function appears in the file *delay_models.c*.
The function calls no other functions.
The function is called by

1. scheduler()

The function reads the global variables

1. line_list[]
2. gate_list[]
3. time
4. next_event

The function assigns values to no global variables.

`void update_value()`

The function appears in the file *event_processor.c*.
The function calls no other functions.
The function is called by

1. event_processor()

The function reads the global variables

1. present_event
2. line_list[]
3. gate_list[]
4. time

The function assigns values to global variables

1. line_list[]
2. gate_event

B.3 Fault simulation

`void activate_faults()`

The function appears in the file *fault_sim.c*.
The function calls macros

1. ZEROUT()
2. SETBIT()

The function is called by

1. fault_sim()

The function reads the global variables

1. line_list[]
2. number_of_lines

The function assigns values to global variables

1. line_list[]

```
void ask_lists()
```

The function appears in the file *options.c*.
The function calls no macros.
The function calls other functions

1. get_line()
2. print_observe_lists()

The function is called by

1. ask_options()

The function reads the global variables

1. number_of_lines
2. trace

The function assigns values to global variables

1. observe_list[]

```
void ask_options()
```

The function appears in the file *options.c*.
The function calls no macros.
The function calls other functions

1. ask_lists()

The function is called by

1. print_det_faults()

The function reads the global variables

1. trace

The function assigns values to no global variables

```
void assign_ip_vector(vector)
char *vector;
```

The function appears in the file *readnew.c*.
The function calls no macros.
The function calls no other functions.
The function is called by

1. read_in()

The function reads the global variables

1. first_pi
2. first_po
3. gate_list[]

The function assigns values to global variables

1. line_list[]

```
int compute_level(gate_index)
int gate_index;
```

The function appears in the file *init.c.*
The function calls no macros.
The function calls no other functions
The function is called by

1. levelize()

The function reads the global variables

1. first_po
2. first_pi
3. gate_list[]
4. line_list[]
5. number_of_lines

The function assigns values to global variables

1. gate_list[]

```
void change_d_db()
```

The function appears in the file *init.c.*
The function calls no macros.
The function calls no other functions
The function is called by

1. fault_sim()

The function reads the global variables

1. line_list[]
2. number_of_lines

The function assigns values to global variables

1. line_list[]

```
void ch_dest_gate_lcount(line_ind, gate_ind)
int line_ind, gate_ind;
```

The function appears in the file *init.c*.
The function calls no macros.
The function calls other functions

1. compute_val()

The function is called by

1. initialize_ckt()

The function reads the global variables

1. gate_list[]
2. line_list[]

The function assigns values to global variables

1. gate_list[]
2. line_list[]

```
void drop_fault(linenum, sa_value)
int linenum, sa_value;
```

The function appears in the file *fault_sim.c*.
The function calls macros

1. ZEROUT()
2. SETBIT()
3. FCOMPLEMENT()
4. FINTERSECT()

The function is called by

1. print_faults()
2. print_det_faults()

The function reads the global variables

1. sim_f_list0
2. sim_f_list1

The function assigns values to global variables

1. sim_f_list0
2. sim_f_list1

```
float fault_coverage(faults_detected)
int faults_detected;
```

The function appears in the file *stats.c*.
The function calls no macros.
The function calls no other functions.
The function is called by

1. print_faults()

The function reads global variables

1. number_of_lines

The function assigns values to no global variables.

```
void fault_sim()
```

The function appears in the file *fault_sim.c*.
The function calls functions

1. change_d_db()
2. activate_faults()
3. levelize()
4. fpropogate()
5. print_det_faults()

The function is called by

1. main() (file *fsim.c*)

The function reads the global variables

1. gate_list[]
2. ckt_depth
3. first_pi

The function assigns values to no global variables

```
void fpropogate(gate)
int gate;
```

The function appears in the file *fault_sim.c*.
The function calls macros

1. FUNION()
2. FCOPY()
3. FINTERSECT()
4. FCOMPLEMENT()

The function calls functions

1. get_f_list()

The function is called by

1. fpropagate()

The function reads the global variables

1. gate_list[]
2. line_list[]
3. fan_table[][]

The function assigns values to global variables

1. line_list[]

`void gen_exhaust_test()`

The function appears in the file *gen_test.c*.
The function calls macros

1. FINTERSECT()
2. SETBIT()
3. BITSET()
4. COPY()

The function calls no other functions.
The function is called by

1. show_step()
2. set_trace_options()

The function reads the global variables

1. first_po
2. first_pi

The function assigns values to no global variables.

`void gen_random_test()`

The function appears in the file *gen_test.c*.
The function calls no macros.
The function calls functions

1. rand_vector()

The function is called by

1. show_step()
2. set_trace_options()

The function reads the global variables

1. first_po

2. first_pi
3. step
4. v_number

The function assigns values to no global variables.

```
void gen_wtd_test()
```

The function appears in the file *gen_test.c*.
The function calls no macros.
The function calls functions

1. rand_vector()

The function is called by

1. show_step()
2. set_trace_options()

The function reads the global variables

1. first_po
2. first_pi
3. step
4. v_number

The function assigns values to no global variables.

```
void get_f_list(gate, f_list_c, f_list_nc)
int gate;
int f_list_c[MAXINTS], f_list_nc[MAXINTS];
```

The function appears in the file *fault_sim.c*.
The function calls macros

1. ZEROUT()
2. FUNION()
3. FCOMPLEMENT()
4. FINTERSECT()

The function is called by

1. fpropagate()

The function reads the global variables

1. gate_list[]
2. line_list[]
3. fan_table[][]

The function assigns values to no global variables.

`void initialize_ckt()`

The function appears in the file *init.c*.
The function calls no macros.
The function calls other functions

1. levelize()
2. ch_dest_gate_lcount()

The function is called by

1. fault_sim()

The function reads the global variables

1. gate_list[]
2. line_list[]
3. first_pi
4. first_po
5. ckt_depth

The function assigns values to global variables

1. gate_list[]
2. line_list[]

`int levelize()`

The function appears in the file *init.c*.
The function calls no macros.
The function calls other functions

1. compute_level()

The function is called by

1. fault_sim()
2. initialize_ckt()

The function reads the global variables

1. first_po
2. first_pi
3. gate_list[]
4. number_of_gates

The function assigns values to global variables

1. gate_list[]

```
void print_det_faults()
```

The function appears in the file *print.c*.
The function calls macros

1. ZEROUT()
2. FINTERSECT()

The function calls other functions

1. drop_fault()
2. ask_options()
3. print_faults()

The function is called by

1. fault_sim()

The function reads the global variables

1. line_list[]
2. gate_list[]
3. number_of_gates
4. faults_to_drop[]

The function assigns values to no global variables.

```
void print_fault_list(fault_vector0, fault_vector1)
int *fault_vector0, *fault_vector1;
```

The function appears in the file *print.c*.
The function calls macros

1. BITSET()

The function calls no other functions
The function is called by

1. fault_sim()
2. print_observe_lists()

The function reads the global variables

1. line_list[]
2. number_of_lines
3. faults_to_drop[]

The function assigns values to no global variables.

```
void print_faults(fault_vector0, fault_vector1)
int *fault_vector0, *fault_vector1;
```

The function appears in the file *print.c*.
The function calls macros

1. BITSET()

The function calls other functions

1. drop_fault()

The function is called by

1. print_det_faults()

The function reads the global variables

1. line_list[]
2. number_of_lines
3. faults_to_drop[]
4. number_detected_faults
5. step
6. trace

The function assigns values to global variables

1. faults_to_drop[]
2. number_detected_faults

```
void print_observe_lists(observe_list)
int observe_list[];
```

The function appears in the file *print.c*.
The function calls macros

1. ZEROUT()
2. FINTERSECT()

The function calls other functions

1. print_fault_list()

The function is called by

1. ask_lists()

The function reads the global variables

1. observe_list[]

The function assigns values to no global variables.

```
char* rand_vector(seed,sig_pr)
int *seed;
float sig_pr[];
```

The function appears in the file *gen_test.c*.
The function calls no macros.
The function calls no other functions.
The function is called by

1. gen_wtd_test()
2. gen_random_test()

The function reads the global variables

1. first_po
2. first_pi

The function assigns values to no global variables.

```
void read_flist(flist)
FILE  *flist;
```

The function appears in the file *read.c*.
The function calls macros

1. SETBIT()

The function calls other functions

1. get_line()

The function is called by

1. main() (file *fsim.c*)

The function reads the global variables

1. number_of_lines

The function assigns values to no global variables.

```
void read_in(test_vector_file)
FILE *test_vector_file;
```

The function appears in the file *readnew.c*.
The function calls no macros.
The function calls other functions

1. assign_ip_vector()

The function is called by

1. main() (file *fsim.c*)

The function reads no global variables.
The function assigns values to no global variables.

```
void read_vector(vector)
char vector[];
```

The function appears in the file *read.c*.
The function calls no macros.
The function calls no other functions.
The function is called by

1. main() (file *fsim.c*)

The function reads the global variables

1. first_pi
2. first_po
3. line_list[]
4. gate_list[]
5. number_test_vectors

The function assigns values to global variables

1. line_list[]

```
void set_trace_options(trace_arg)
char * trace_arg;
```

The function appears in the file *show_step.c*.
The function calls no macros.
The function calls other functions

1. gen_exhaust_test()
2. gen_random_test()
3. gen_wtd_test()

The function is called by

1. main() (file *fsim.c*)

The function reads global variables

1. trace

The function assigns values to global variables

1. vector_file

```
void show_step()
```

The function appears in the file *show_step.c*.
The function calls no macros.
The function calls other functions

1. gen_exhaust_test()
2. gen_random_test()
3. gen_wtd_test()

The function is called by

1. main() (file *fsim.c*)

The function reads global variables

1. step

The function assigns values to global variables

1. vector_file

```
void sig_prob_query(sig_pr)
float sig_pr[];
```

The function appears in the file *show_step.c*.
The function calls no macros.
The function calls no other functions.
The function is called by

1. gen_wtd_test()

The function reads global variables

1. trace
2. first_pi
3. first_po
4. gate_list[]

The function assigns values to global variables

1. line_list[]
2. sig_pr[]