HITEC/PROOFS USER'S MANUAL

Center for Reliable & High-Performance Computing
University of Illinois
1308 West Main Street
Urbana, IL 61801-2307

January 11, 1996

HITEC/PROOFS is a gate-level, sequential circuit test generation and fault simulation package which targets single stuck-at faults in synchronous sequential circuits described in the ISCAS89 benchmark format. Fault lists are automatically generated for all single stuck-at faults in a circuit, and fault collapsing is performed using structural equivalence. Individual faults are targeted by HITEC, and several passes through the fault list are made, with increasing time and backtrack limits in each successive pass. When a test sequence is successfully generated to detect a target fault, HITEC invokes the PROOFS fault simulator to find all additional faults incidentally detected by the test sequence. Detected faults are then removed from the fault list. After each pass through the fault list, the user is prompted about whether to continue with the next pass; execution terminates when the user responds negatively. PROOFS can also be run separately using a test set supplied by the user.

The HITEC/PROOFS package consists of two main programs and four preprocessing programs. The two main programs are the test generator and the fault simulator. The preprocessing steps were implemented as separate programs for several reasons. First, memory overhead is a major concern for both the test generator and fault simulator; therefore, by separating the programs, a minimal amount of memory is needed by the test generator. Second, it allows the user to write a parser or fault list creator without looking at the internal data structures. The four preprocessing programs levelize the circuit, create a complete fault list, collapse the fault list, and calculate the dominators in a circuit.

I System Overview

Figure 1 shows the flow diagram for the use of HITEC or PROOFS. All run-time parameters are stored in the file *TEST.run*. This was done to avoid typing command line arguments and to allow easier running by system calls from a graphical interface. In the flow diagram,

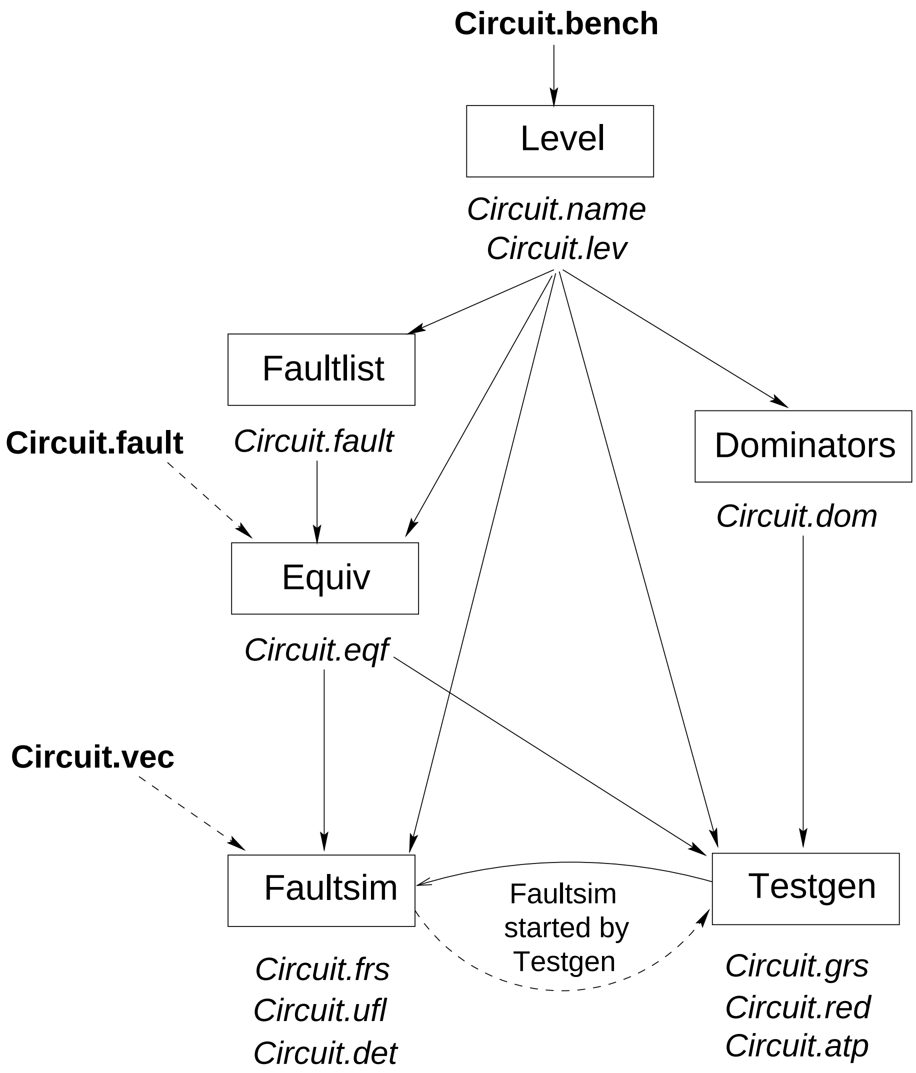


Figure 1: Run time flow diagram

the user can either supply a fault list or create a complete fault list. In addition, the user can supply functional vectors to evaluate before test generation is begun. The names in boxes are the commands, the file names in italics are created by a program, and the bold names are files supplied by the user.

II Parsing

The parsing program, *level*, was written in C and lex. It reads the flattened circuit description of the ISCAS benchmark format. The benchmark file for an example circuit, s27, is shown in Figure 2, and the circuit schematic is shown in Figure 3.

```
INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)
OUTPUT(G17)
G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)
G14 = NOT(G0)
G17 = NOT(G11)
G8 = AND(G14, G6)
G15 = OR(G12, G8)
G16 = OR(G3, G8)
G9 = NAND(G16, G15)
G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
END
```

Figure 2: Benchmark file for s27 (s27.bench)

Aside from parsing the circuit, *level* levelizes the circuit, flags asynchronous feedback, and calculates the testability measures for the circuit. A tokenized format of the circuit is then written out to the *circuit*.**lev** file. Identifiers are assigned to the nodes in levelized order, as shown in the revised circuit schematic in Figure 4. A net name to net number translation table is written to the *circuit*.**name** file. An example for s27 is shown in Figure 5.

The format of the *circuit*.**lev** file for s27 is shown in Figure 6. The number on the first line is the number of gates in the circuit plus one; the second number is obsolete. Each line starting with the third line represents one gate. The first number is the node identifier. The second number is the token for the gate type, where the gate type tokens are specified in

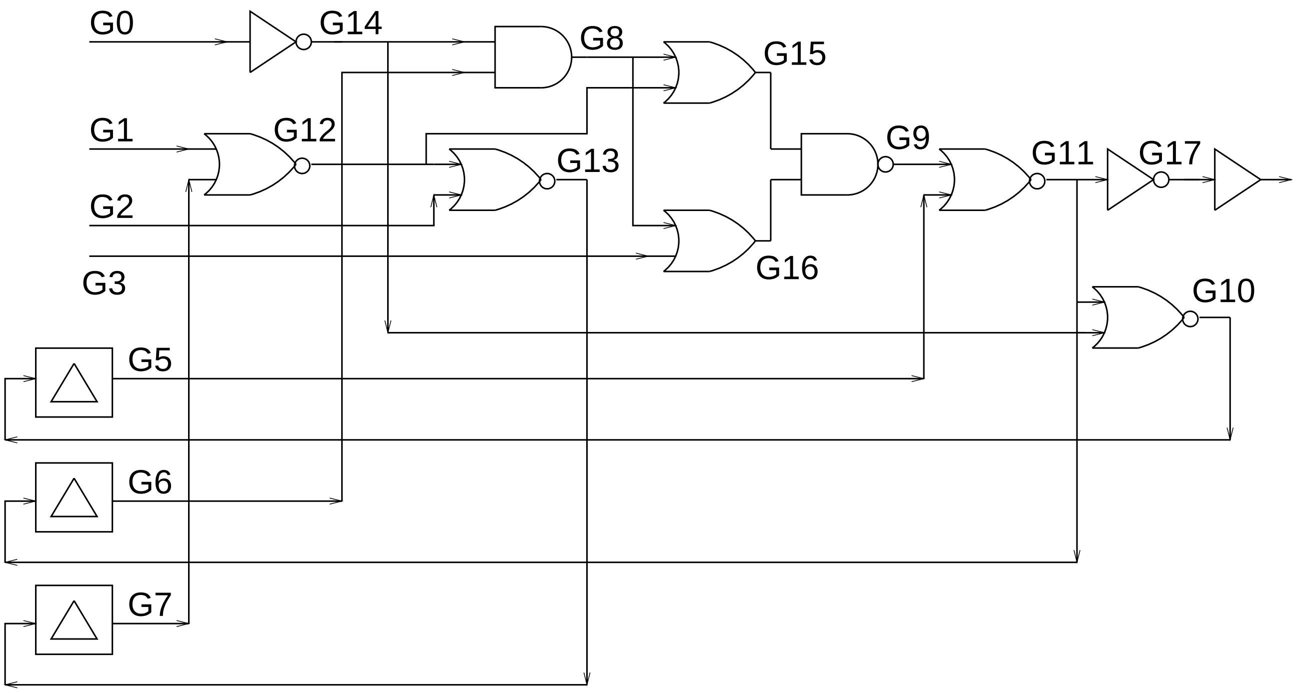


Figure 3: s27 circuit schematic

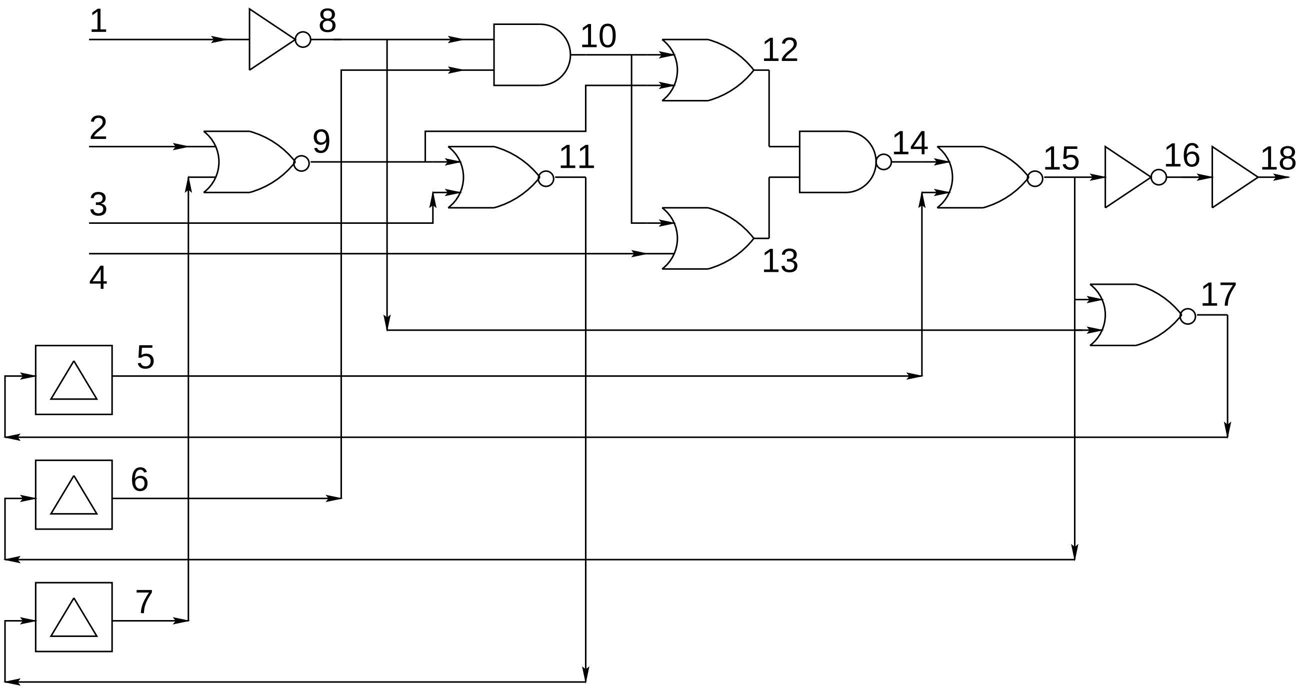


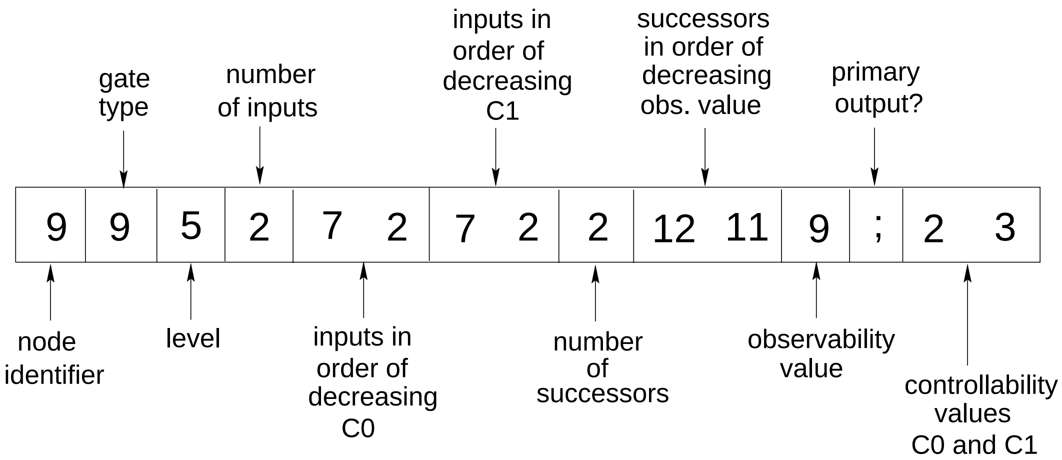
Figure 4: s27 circuit schematic with levelized identifiers

```
1 G0
2 G1
3 G2
4 G3
5 G5
6 G6
7 G7
8 G14
9 G12
10 G8
11 G13
12 G15
13 G16
14 G9
15 G11
16 G17
17 G10
18 G17_$OUTPUT
```

Figure 5: Net name to net number translation for s27 (s27.name)

Table I. The third number is the level of this gate in the circuit. The fourth number is the number of inputs to this gate. Next is the list of input lines to this gate sorted in order of decreasing (easier to control) values of controllability zero. Next is the list of input lines to this gate sorted in order of decreasing (easier to control) values of controllability one. Next is the number of successors of this gate, followed by the list of successor gates sorted in order of decreasing observability values (easier to observe). The next number is the observability of this line. The next character is a semicolon or an O. If there is an O, then this line is a primary output. The last two numbers are the values of controllability zero and one. This structure was chosen to allow zero look-head parse of the description. This structure allows all arrays to be dynamically allocated to their exact needed size while the description is being read.

Levelization is done by setting all primary inputs and flip-flops to level 0, and performing an event-driven calculation of the level of each gate in the circuit. Any gate with an unassigned level is in an asynchronous feedback loop or is a successor of an unconnected line.




```
19
10
1 1 0 0 1 8 16 ; 0 0
2 1 0 0 1 9 11 ; 0 0
3 1 0 0 1 11 13 ; 0 0
4 1 0 0 1 13 12 ; 0 0
5 5 0 1 17 17 1 15 7 ; 2 10
6 5 0 1 15 15 1 10 10 ; 8 11
7 5 0 1 11 11 1 9 10 ; 1 3
8 10 5 1 1 1 2 10 17 16 ; 1 1
9 9 5 2 7 2 7 2 2 12 11 9 ; 2 3
10 6 10 2 6 8 6 8 2 12 13 8 ; 3 14
11 9 10 2 9 3 9 3 1 7 10 ; 1 3
12 8 15 2 10 9 10 9 1 14 5 ; 6 4
13 8 15 2 10 4 10 4 1 14 8 ; 4 1
14 7 20 2 12 13 12 13 1 15 3 ; 6 5
15 9 25 2 14 5 5 14 3 6 17 16 0 ; 8 11
16 10 30 1 15 15 1 18 0 ; 11 8
17 9 30 2 15 8 15 8 1 5 7 ; 2 10
18 2 35 1 16 16 0 0 O 11 8
```

Figure 6: Levelized circuit description for s27 (s27.lev)

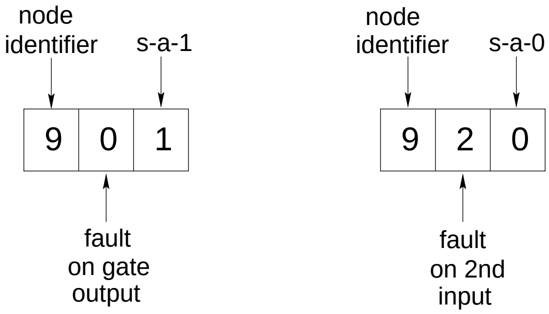
Table I: TOKEN-TO-GATE-TYPE TRANSLATION

Token	**Gate Type**	**Token**	**Gate Type**
1	INPUT	21	TRISTATE
2	OUTPUT	22	TRISTATEINV
3	XOR	23	TRISTATE1
4	XNOR	24	DFF_L
5	DFF	25	DFF_LCP
6	AND	26	DFF_LC
7	NAND	27	DFF_LP
8	OR	28	DFF_CP
9	NOR	29	DFF_C
10	NOT	30	DFF_P
11	BUF	31	NAND_LATCH
12	TIE1	32	NOR_LATCH
13	TIE0	33	CMOS
14	TIEX	34	NMOS
15	TIEZ	35	PMOS
16	MUX_2	36	NOTIF1
17	BUS	37	NOTIF0
18	BUS_GOHIGH	38	BUFIF1
19	BUS_GOLOW	39	BUFIF0
20	BUSZ		

The testability measures are calculated using the SCOAP testability measurement technique. The measurements are calculated through the flip-flops, and calculation continues until there is convergence.

III Fault Lists

The program *faultlist* will create a complete, uncollapsed fault list for the circuit in the *circuit*.**fault** file. If the user wishes to supply a fault list, then he or she may create a fault list in the same format and supply it to the fault collapser. The format of the fault list is shown in Figure 7. Each line represents one fault and ends in a semicolon. The first number is the gate on which the fault is present. The second number is the input number of the fault, where input number 0 is the output of the gate. The third number is the fault type, i.e., 0 or 1 for a stuck-at-0 or stuck-at-1 fault. For example, the fault *15 2 0* is a stuck-at-0 fault on the second input of gate 15.



$\vdots$
7 0 0;
7 0 1;
7 1 0;
7 1 1;
8 0 0;
8 0 1;
8 1 0;
8 1 1;
9 0 0;
9 0 1;
9 1 0;
9 1 1;
9 2 0;
9 2 1;
$\vdots$

Figure 7: Fault list for s27 (s27.fault)

The program *equiv* will collapse any fault list and also order the faults in one of four ways. The faults are collapsed using a few simple rules shown in Table II. The fault groups are then ordered in either a random, breadth-first, depth-first from the primary inputs order, or a depth-first from the primary outputs order. The depth-first from the primary outputs order was shown to give the best performance for fault simulation. A portion of the equivalent fault file for the s27 circuit is shown in Figure 8. Equivalent faults are listed on the same line, separated by colons, and the fault closest to the primary outputs is listed first. This fault is used as the representative fault for the fault group by the test generator and fault simulator.

Table II: FAULT COLLAPSING RULES

Gate	**Fault**	**Fault Collapsed Into**
NOT	A s-a-0	Z s-a-1
NOT	A s-a-1	Z s-a-0
AND	A s-a-0	Z s-a-0
NAND	A s-a-0	Z s-a-1
OR	A s-a-1	Z s-a-1
NOR	A s-a-1	Z s-a-0

A is an input to the gate.
Z is the output of the gate.

```
   ⋮
11 1 0 ;
12 1 0 ;
18 0 0 : 18 1 0 : 16 0 0 : 16 1 1 ;
15 0 1 ;
15 2 0 : 5 0 0 ;
5 1 0 : 17 0 0 : 17 2 1 : 17 1 1 ;
8 0 1 : 8 1 0 : 1 0 0 ;
15 1 0 : 14 0 0 ;
14 2 1 : 13 0 1 : 13 2 1 : 4 0 1 : 13 1 1 ;
10 0 1 ;
   ⋮
```

Figure 8: Equivalent fault file for s27 (s27.eqf)

IV Dominators

The program *dominators* is used to calculate the static dominators of each node in the circuit and to determine all the mandatory assignments to propagate a D or D bar on the input of a given gate. In the current implementation, the dominators are restricted to one time frame. In other words, dominators crossing a time frame are not computed. The dominators and mandatory assignments are determined in reverse level order, i.e., from primary outputs to primary inputs. Figure 9 is the algorithm to determine the dominator gates for each gate in the circuit and the algorithm to determine all mandatory assignments needed for a gate. There are two simple rules to determine the dominators for a given gate. First, the gate itself is a dominator. Second, the intersection of the dominators of all the successor gates also contains dominators to the gate.

For **level** = maxlev downto 0
    For each gate **G** in **level**
$$DOM(G) \ = \ G \cup (DOM(Succ\ 1\ of\ G) \cap DOM(Succ\ 2\ of\ G) \cap DOM(Succ\ 3\ of\ G) \cdots)$$

For each gate **G**
    $MAN(G) = \phi$
    For each D in DOM(**G**)
$$MAN(G) = MAN(G) \cup (All\ off\ path\ inputs\ of\ D)$$

Figure 9: Algorithm to determine dominators and mandatory assignments

The parity of the dominator gate is also retained to determine if the mandatory assignment affects the good or the faulty machine during test generation. Figure 10 is an example of how the parities are used to determine mandatory assignments. In this example, the value **1/0** is the value on the input of the NAND gate. Only the good value of this gate needs to be set to the noncontrolling value so that the off-path input of gate A is assigned the value **1/X**. Gate B is a dominator of gate A with odd parity, because gate A is an inverting gate. The odd parity means that the D value will be inverted an odd number of times before reaching the input of gate B; therefore, only a faulty value needs to be set, so that the off-path gate inputs have the mandatory assignments of **X/1**. In a similar way, gates C and D have even parity; therefore, the off-path lines have a mandatory assignment of **X/0**.

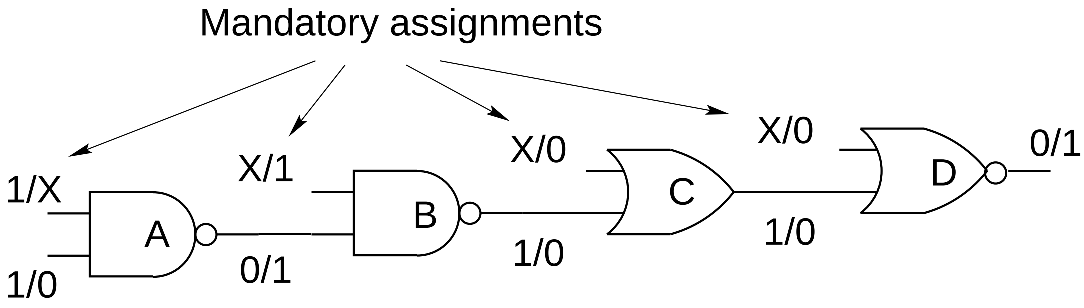


Figure 10: Determination of dominators and mandatory assignments

The dominator file of s27 is shown in Figure 11. The format of the file consists of one line for each gate. The first number represents the gate for which the mandatory assignments correspond. The second number is the number of mandatory assignments. Each mandatory assignment is made up of three numbers: the gate, the off-path line number predecessor to this gate, and the parity of the gate.

```
1 0 ;
2 1 9 7 0;
3 1 11 9 0;
4 3 15 5 1 14 12 0 13 10 0;
5 2 15 14 0 5 0 0;
6 3 15 5 1 10 8 0 6 0 0;
7 2 9 2 0 7 0 0;
8 1 8 1 0;
9 2 9 7 0 9 2 0;
10 3 15 5 1 10 6 0 10 8 0;
11 2 11 9 0 11 3 0;
12 4 15 5 1 14 13 0 12 10 0 12 9 0;
13 4 15 5 1 14 12 0 13 10 0 13 4 0;
14 3 15 5 0 14 12 0 14 13 0;
15 2 15 14 0 15 5 0;
16 1 16 15 0;
17 2 17 15 0 17 8 0;
18 1 18 16 0;
END
```

Figure 11: Dominators of s27 (s27.dom)

V Fault Simulation and Test Generation

The fault simulator can be run as a stand-alone program to fault grade a set of test vectors, or it can be run in conjunction with the test generator to identify all faults detected by the test sequences generated. When run as a stand-alone program, the fault simulator takes a test vector file as input, *circuit***.vec**, in addition to the *circuit***.lev** and *circuit***.eqf** files. A test vector file for the s27 circuit is shown in Figure 12. The order of primary inputs in

```
4
0010
1001
1100
0110
0011
0111
1001
1011
0011
0011
0000
1110
1001
0010
1000
1010
0010
0101
0101
1001
END
```

Figure 12: Test vector file for s27 (s27.vec or s27.atp)

the *circuit***.vec** file is the same as that in the *circuit***.lev** and *circuit***.name** files. The fault simulation results are stored in the *circuit***.frs** file which contains the run time, fault coverage, and other statistics about the fault simulation. The fault simulator also creates a *circuit***.ufl** file containing all undetected faults and a *circuit***.det** file containing all detected faults, both in the same format as the collapsed fault list.

The test generator automatically invokes the fault simulator and communicates with it through UNIX sockets. The test generation results are stored in the *circuit***.grs** file which contains the test generation run time, test generation efficiency, and other statistics. The resulting test set is stored in the *circuit***.atp** file, which has the same format as the *circuit***.vec** file, as shown in Figure 12. Untestable faults are stored in the *circuit***.red** file. The user can supply an initial set of test vectors in a *circuit***.vec** file to reduce the number of faults targeted by the test generator. The *TEST.run* file must be configured to enable this option. The *TEST.run* file format is shown in Figure 13.

simpic port number	- port number for faultsim graphics (obsolete)
master host name	- name of master host for graphics (obsolete)
bench file	- ckt.bench (name of benchmark file – level input)
lev file	- ckt.lev (name of levelized circuit file – level output)
vec file	- ckt.vec (name of vector file – faultsim input)
fault file	- ckt.fault (name of fault file – faultlist output)
equiv fault file	- ckt.eqf (name of equivalent fault file – equiv output)
undetected fault file	- ckt.ufl (name of undetected fault file – faultsim output)
faultsim host name	- used with testgen (name of host for faultsim)
testgen host name	- used with testgen (name of host for testgen)
name file	- ckt.name (name of "name" file – level output)
atp file	- ckt.atp (test vector file – testgen output)
faultsim result file	- ckt.frs (faultsim results)
testgen result file	- ckt.grs (testgen results)
sim gen port	- port between faultsim and testgen
color	- color graphics used if one (obsolete)
faultsim running	- faultsim running if one
testgen running	- testgen running if one
fault pic running	- graphics running if one (obsolete)
read vec	- read ckt.vec file before test generation if one
dominator file	- ckt.dom (name of dominators file – dominators output)
redundant fault file	- ckt.red (name of untestable fault file – testgen output)
debug	- debug value for testgen
backtrack limit	- maximum number of backtracks allowed in testgen
state backtrack limit	- maximum number of state backtracks allowed in testgen
time limit	- maximum time allowed per fault in pass 1 (unit = 1/100 sec for HP, 1/60 sec for SUN)
scan stat file	- name of file for scan statistics (obsolete)

Figure 13: TEST.run file format

VI Running HITEC/PROOFS

To run the test generator and fault simulator, first place all executables in a *bin* directory, and include the pathname of that directory in your search path. Then create a *TEST.run* file with one of the two script files provided, *do_hitec* or *do_proofs*; *do_hitec* configures the *TEST.run* file with options set for running the test generator (which invokes the fault simulator), and *do_proofs* configures the *TEST.run* file with options set for running the fault simulator only. All programs except for *level* look at the *TEST.run* file for their arguments. Next, run the preprocessing programs, followed by the fault simulator or test generator. Program invocations for the s27 circuit are as follows:

do_hitec s27	(or *do_proofs s27*)
level s27	
faultlist	
equiv	
dominators	(not needed if the test generator is not used)
testgen	(or *faultsim*)

The following component types are handled by HITEC and PROOFS:

INPUT	
OUTPUT	
DFF	– D flip flop
AND	
NAND	
OR	
NOR	
NOT	– inverter
BUF	– buffer, output = input
TIE1	– line tied to 1
TIE0	– line tied to 0
TIEX	– line tied to X
TIEZ	– line tied to Z

In addition, PROOFS handles the following component types also:

XOR	– exclusive-OR
XNOR	– exclusive-NOR

BUS	– bus, output goes to unknown if all inputs have Z value
BUS_GOHIGH	– bus, output goes to high if all inputs have Z value
BUS_GOLOW	– bus, output goes to low if all inputs have Z value
TRISTATE	– tristate, active low control signal
TRISTATEINV	– tristate with output inverted
TRISTATE1	– tristate, active high control signal
MUX2	– 2-input multiplexor

For the MUX2, TRISTATE, TRISTATEINV, and TRISTATE1 components, the control signal must be the *last* input in the list of inputs. In the multiplexor, the first input is selected when the control signal is 0, and the second input is selected when the control signal is 1. A bus component must have only tristate components as predecessors, and tristate components must have only bus components as successors. The bus output value is the value of the non-Z (high impedance) input. If there is a conflict (a one placed on one bus input and a zero placed on another bus input), the bus output value is unknown.