A Modified Parallel Fault Simulator for Combinational Circuits

Shih-Ming Li

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Electrical and Computer Engineering Commons

Recommended Citation
https://scholarworks.wmich.edu/masters_theses/987
A MODIFIED PARALLEL FAULT SIMULATOR
FOR COMBINATIONAL CIRCUITS

by
Shih-Ming Li

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Electric Engineering

Western Michigan University
Kalamazoo, Michigan
April 1991
A MODIFIED PARALLEL FAULT SIMULATOR
FOR COMBINATIONAL CIRCUITS

Shih-Ming Li, M.S.
Western Michigan University, 1991

The most commonly used model for fault analysis is called the "Stuck-At" model. With this model, a faulty gate input and/or output is modeled as Stuck-at-0 (s-a-0) or Stuck-at-1 (s-a-1). After a certain number of test vectors are applied to a network, the percentage fault coverage is computed. This computed value relates to the percentage of stuck-at faults detectable at the output. Various studies have been made to enhance the performance of fault simulators. In this paper, a modified parallel simulator MODPAR for combinational circuits is presented. A comparison of simulation results from MODPAR is made with that of SCIRTSS (Hill & Huey, 1977), which employs a parallel simulation algorithm. Simulation data indicate that MODPAR enjoys simulation time advantage.
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Abuelyaman for his guidance throughout this study. I would also like to express deep appreciation to the many people who have been involved in the preparation of this study.

Last, I would like to dedicate this thesis to my girlfriend, Sheau-Ping Shyy, whose support made this endeavor worthwhile.

Shih-Ming Li
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A modified parallel fault simulator for combinatorial circuits

Li, Shih-Ming, M.S.
Western Michigan University, 1991
NOTE TO USERS

THE ORIGINAL DOCUMENT RECEIVED BY U.M.I. CONTAINED PAGES WITH POOR PRINT. PAGES WERE FILMED AS RECEIVED.

THIS REPRODUCTION IS THE BEST AVAILABLE COPY.
TABLE OF CONTENTS

ACKNOWLEDGEMENTS.................................................. ii
LIST OF TABLES......................................................... iv
LIST OF FIGURES........................................................ v

CHAPTER

I. INTRODUCTION....................................................... 1

II. REVIEW OF FAULT SIMULATION ALGORITHMS......................... 3
    Parallel Simulation.............................................. 3
    Deductive Simulation.......................................... 5
    Concurrent Simulation......................................... 7
    Comparison and Choice...................................... 9

III. IMPLEMENTATION.................................................. 12
    Input File and Data Structure................................ 12
    Fault Collapsing and Classification.......................... 14
    Testing Decisions............................................. 16

IV. SIMULATION RESULTS............................................... 18
    Tested Circuits................................................. 18
    Summary of Results........................................... 18

V. FUTURE IMPROVEMENT................................................ 20

APPENDICES

A. FORTRAN Version of Programs
   "MODPAR" and "CONVERT".......................................... 21

B. HPS Files for Tested Circuits.................................. 38

BIBLIOGRAPHY.......................................................... 43

iii
LIST OF TABLES

1. Review of Stuck-at Fault Concept ..................... 4
2. Comparison Between Deductive and Concurrent Simulation .................. 8
3. Data Format of Stage_3 Output ........................ 13
4. New Data Structure of CONVERT .......................... 13
5. Circuit Description and Numbers of Total and Collapsed Faults for Bench Mark Circuits ........................ 19
6. Simulation Data for SCIRTSS and MODPAR ............. 19
LIST OF FIGURES

1. Example Circuit and Fault Words ................... 3
2. Concurrent Simulation of a NAND Gate .......... 10
3. I/O Fault Classification for an AND Gate .... 15
4. Algorithm for MODPAR ............................ 17

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER I

INTRODUCTION

The performance of fault simulation algorithms is of vital importance since it might take days, or even months of CPU time to simulate a VLSI circuit. A circuit size is directly proportional to its corresponding fault lists and input test vectors. In the worst case, the simulation time may increase in proportion to the third power of circuit size (Breuer & Friedman, 1976).

Various fault simulation algorithms have been developed to respond to the rapidly escalating CPU time. Most of the developed techniques fall into one of three leading techniques: parallel, deductive, and concurrent.

The merits of each of these techniques will be reviewed in Chapter II. Each will be judged in terms of the simulation process, data storage, and the number of passes required. We will show that parallel simulation algorithm, which ranks third for large circuits, could be improved. Our strategy is to study its disadvantages and eliminate them. In the process, a new fault simulation algorithm MODPAR will be introduced. This algorithm will be interfaced with the STAGE_3 compiler of SCIRTSS (Hill & Huey,
1977), which has been re-implemented in a user-friendly form to handle VLSI circuits. The parallel simulator of SCIRTSS will be used as speed bench mark for its accuracy.

In Chapter III, the implementation of MODPAR and two types of fault collapsing will be introduced. Simulation results will be summarized in Chapter IV. Finally, future improvements will be discussed in Chapter V.
CHAPTER II

REVIEW OF FAULT SIMULATION ALGORITHMS

Parallel Simulation

Consider a host machine with word size \( W \), then \( W \) different problems (i.e., \( W-1 \) faults and one bit for the fault-free machine) can be simulated in parallel. An example for parallel simulation is given for the circuit in Figure 1. The definitions of all stuck-at faults for the circuit in Figure 1 are included in Table 1.

\[
\begin{align*}
\text{a: } & \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \\
\text{b: } & \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\
\text{c: } & \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \\
\text{d: } & \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \\
\text{e: } & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \\
\text{f: } & \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}
\end{align*}
\]

Figure 1. Example Circuit and Fault Words.
Bit 0 of each fault word represents the value of the fault-free output, while each stuck-at fault in the circuit is assigned to one bit position in the fault word (Miczo, 1986). If the input test vector \( \{ a,b,c \} = \{ 1,0,1 \} \) is applied to node a, then the fault word will be \([11111, 101]\). This shows that the value of this node is 1, except for the case where the node a is s-a-0. Faults must be injected into the simulator in such a way that any individual fault affects only one bit position. This is accomplished by a process called "bugging the simulator" (Miczo, 1986). The faulty value (s-a-1 or s-a-0) replaces the corresponding bit value in the fault word after the evaluation of node output. When a pass through the simulator is completed, the observable outputs are checked by the control program. Values in any bit position which differ from

<table>
<thead>
<tr>
<th>Number</th>
<th>Node</th>
<th>Fault Type</th>
<th>Number</th>
<th>Node</th>
<th>Fault Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>a</td>
<td>s-a-0</td>
<td>(2)</td>
<td>a</td>
<td>s-a-1</td>
</tr>
<tr>
<td>(3)</td>
<td>b</td>
<td>s-a-0</td>
<td>(4)</td>
<td>b</td>
<td>s-a-1</td>
</tr>
<tr>
<td>(5)</td>
<td>c</td>
<td>s-a-0</td>
<td>(6)</td>
<td>c</td>
<td>s-a-1</td>
</tr>
<tr>
<td>(7)</td>
<td>d</td>
<td>s-a-0</td>
<td>(8)</td>
<td>d</td>
<td>s-a-1</td>
</tr>
<tr>
<td>(9)</td>
<td>e</td>
<td>s-a-0</td>
<td>(10)</td>
<td>e</td>
<td>s-a-1</td>
</tr>
<tr>
<td>(11)</td>
<td>f</td>
<td>s-a-0</td>
<td>(12)</td>
<td>f</td>
<td>s-a-1</td>
</tr>
</tbody>
</table>

Table 1
Review of Stuck-at Fault Concept
the good machine output (bit 0) indicate that the faults associated with these bit positions are detected. In the above example, faults 5, 10, and 11 can be detected at the output with input \(1,0,1\).

Now consider a circuit with \(N\) s-a faults and a host machine of word size \(W\); it will take \([N/W-1]\) passes to simulate the circuit with one test vector (Chang, Chappell, Elmendorf, & Schmidt, 1974). For LSI or VLSI circuits, the efficiency of parallel simulation will be degraded due to the multi-pass requirements. However, parallel simulation utilizes a predefined storage space and uses bit-level logic operations which guarantees running time superiority.

**Deductive Simulation**

Deductive simulation (Armstrong, 1972) is based on the concept that the simulator deduces which faults are tested by an input vector and creates lists of those that are sensitized at any given node (Miczo, 1986). As simulation proceeds, the effect of some faults will disappear, hence, these faults are dropped by the simulator. The following is the algorithm of deductive simulation (Breuer & Friedman, 1976):
LET $E_i(x_1, x_2, \ldots, x_n)$ be a Boolean Expression for node $i$ in terms of inputs $x_1, x_2, \ldots, x_n$, and $L_j$ be the fault list associated with node $j$.

FOR Each node in sequence $k$

IF Value of $E_k$ is 0

THEN Replace all $x_i$'s ($\overline{x_i}$'s) in $E_k$ by $L_{x_i}$'s ($\overline{L_{x_i}}$'s)

ELSE Replace all $x_i$'s ($\overline{x_i}$'s) in $E_k$ by $\overline{L_{x_i}}$'s ($L_{\overline{x_i}}$'s)

ENDIF

Replace '$\land$' in $E_k$ by '$\lor$' and '$\lor$' by '$\land$'

Add $k_0$ ($k_1$) to $L_k$ if value of $E_k$ is 1 (0)

ENDFOR;

To illustrate this procedure, consider the circuit in Figure 1.

$L_a = \{a_0\} ; L_b = \{b_1\} ; L_c = \{c_0\}$

Thus,

$L_d = L_a \cap L_b \cup d_0 = \{b_1, d_0\}$

$L_e = \overline{L_b} \cap L_c \cup e_1 = \{c_0, e_1\}$

$L_f = \overline{L_d} \cap L_e \cup f_0 = \{c_0, e_1, f_0\}$

Note that when $L_j$ is computed, the new $L_j$ must be compared with the old $L_j$, before the latter is destroyed. In this sense, when applying another test vector, all lines need not be reevaluated (Breuer & Friedman, 1976).

As stated in the above algorithm, deductive simulation requires the computation of set unions and intersections, which take most of the CPU time used. A more sophisticated way for data storage is needed because of the unpredictable size of fault lists.
Concurrent Simulation

For deductive simulation, the equations for fault list manipulation are applied to get the fault list at the corresponding node output even though no logic activity has occurred at the output of the fault free gate.

The concurrent simulation (Ulrich & Baker, 1974) avoids set union, intersection and difference calculation by appending or linking new copies of circuit elements to the original elements whenever faults cause faulted machine signals to differ from good machine signals (Miczo, 1986). Furthermore, new circuit elements are added as long as the error signals continue to propagate.

Again, consider the circuit in Figure 1 for illustration. All the circuit elements along with each node are compared with those of deductive simulation and are listed in Table 2. For the first NAND gate, the values of the I/O are (1 0 1). Therefore, three sets of SFLs are included:

1. a s-a-0 ; 0 0 1
2. b s-a-1 ; 1 1 0
3. d s-a-0 ; 1 0 0

The good machine has the I/O value of (1 0 1). As observed from the SFLs, faults 2 and 3, which have different output values compared with the good machine, can be detected.
### Table 2
Comparison Between Deductive and Concurrent Simulation

<table>
<thead>
<tr>
<th>Node</th>
<th>Deductive Simulation (Fault List)</th>
<th>Concurrent Simulation (Super Fault List)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>(5) c s-a-0</td>
<td>(4) b s-a-1 (1 0/0)</td>
</tr>
<tr>
<td></td>
<td>(10) e s-a-1</td>
<td>(5) c s-a-0 (0 0/1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(10) e s-a-1 (0 1/1)</td>
</tr>
<tr>
<td>d</td>
<td>(4) b s-a-1</td>
<td>(1) a s-a-0 (0 0/1)</td>
</tr>
<tr>
<td></td>
<td>(7) d s-a-0</td>
<td>(4) b s-a-1 (1 1/0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(7) d s-a-0 (1 0/0)</td>
</tr>
<tr>
<td>f</td>
<td>(5) c s-a-0</td>
<td>(4) b s-a-1 (0 0/1)</td>
</tr>
<tr>
<td></td>
<td>(10) e s-a-1</td>
<td>(7) d s-a-0 (0 0/1)</td>
</tr>
<tr>
<td></td>
<td>(11) f s-a-0</td>
<td>(5) c s-a-0 (1 0/0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(10) e s-a-1 (1 1/0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(11) f s-a-0 (1 0/0)</td>
</tr>
</tbody>
</table>

The concurrent simulation considers all faults in one pass as does the deductive simulation. When applying another test vector, only those elements which do not agree with the same elements in the fault-free circuit are explicitly simulated (Breuer & Friedman, 1976). Hence the concurrent simulation can potentially lead to a reduction in CPU time. However, more storage is needed than in deductive simulation.
Comparison and Choice

Several studies have been published which compare the efficiency of various simulation techniques. A comparison between parallel and deductive simulation indicates that, in general, the deductive method is better for a wide range of circuit size and complexity (Chang, Chappell, Elmendorf, & Schmidt, 1974). Parallel simulation is better only in simulating small highly sequential circuits (e.g., < 500 gates). Another study confirms that deductive simulation is faster than parallel simulation (Ozguner, 1980).

Deductive and concurrent simulations enjoy advantages over parallel simulation in that when large portions of faults are detected, the former ones require progressively less execution time to detect new faults (Abuelyaman, 1988). This is because simulation will restart from the primary inputs every time it detects no activity at some node input and/or output. For parallel simulation, however, the testing process starts from the primary inputs, and the lists of detected faults are examined at the primary output. This process involves many redundant tasks. Further research efforts have been targeted to concurrent simulation because of its efficiency (Krohn, 1981; Ulrich, 1980).

Illustration of the difference between these simulation algorithms is given in Figure 2 using a NAND gate.
For concurrent simulation, super fault lists (SFLs) are given along with each node. If there aren't any new faults detected for a test vector, then the path will halt there. In Figure 2, the previous input to the NAND gate is (1 0), and the data in the super fault lists show that input b s-a-1 and output c s-a-0 are detectable at the node output. If the input changes to (0 0), there won't be any new element added to the detectable fault lists because input a s-a-1 is not detectable. Since the output under fault (7) is also 0 (the same as for input (1 0)), simulation through this path will then halt at the next gate. If all the paths in the circuit are terminated, another input test vector will be applied to the circuit instead of testing all the way to the primary outputs.
Therefore, many unnecessary simulation passes and CPU time are needed to detect some hidden faults using parallel simulation. Suggestions are made in this study to reduce the unnecessary simulation passes.

On the other hand, instead of dealing with set unions and intersections in deductive simulation and scanning process in concurrent simulation, parallel simulation employs direct logic operations of node inputs to compute the node output. The main difference is that parallel simulation employs memory-to-memory, bitwise logic operation for each node, while the former ones employ higher level operations throughout the simulation process.

Since faults are divided into several words ($\lceil N/(W-1) \rceil$ words), comparison can be made among different fault partitioning techniques to further enhance the efficiency of parallel simulation. Henceforth, our discussion will be based on the parallel algorithm.
CHAPTER III

IMPLEMENTATION

A new fault simulation technique is written which utilizes parallel simulation algorithms. The implementation of this simulator is discussed as follows.

Input File and Data Structure

In SCIRTSS (Hill & Huey, 1977), information about the circuit to be tested is written in hardware transfer language. After three stages of compilation, SCIRTSS links the STAGE_3 circuit description and the parallel fault simulator using a subroutine called "Fmndlst". The data structure of the STAGE_3 output file is given in Table 3. The circuit description given by the STAGE_3 output file includes node type, circuit processing sequence, and the fan-in node numbers.

MODPAR starts from the circuit description of the STAGE_3 and reconstructs a new data structure using a program CONVERT. The output file of this program gives a new circuit description in a form shown in Table 4. The first element in the array denotes the sequence of a node, and the second element indicates the node type which inc-
Table 3

Data Format of Stage_3 Output

<table>
<thead>
<tr>
<th>Gate Type</th>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
<th>Node 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND</td>
<td>11003</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>2</td>
<td>11001</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>3</td>
<td>11004</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NAND</td>
<td>11001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NAND</td>
<td>11003</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

* The first column defines the gate type and also the sequence; the remaining columns are the fan-in node numbers.

Table 4

New Data Structure of CONVERT

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Gate Type</th>
<th>Fan-In</th>
<th>Fan-Out</th>
<th>Fan-In</th>
<th>Fan-Out</th>
<th>Fan-In</th>
<th>Fan-Out</th>
<th>Fan-In</th>
<th>Fan-Out</th>
<th>Fan-In</th>
<th>Fan-Out</th>
<th>Fan-In</th>
<th>Fan-Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>24</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>13</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

* The first column defines the sequence; the second defines the gate type. The third and tenth columns define the fan-in and fan-out numbers. Fan-in and fan-out numbers reside at columns 4 to 9 and 11 to 17 separately. Column 17 is reserved for node value.

Induces PI, PO, and 5 basic gate types (ie., AND, NAND, OR, NOR, XOR). The remaining entries in the array describe the
fan-in and fan-out connections, and the last element is reserved for the value associated with each node. For example, if a row in the array is given by the following pattern, \([7,0,0,0,0,0,0,0,3,12,13,14,0,0,0,0]\), then one can see that the pattern represents the seventh node in sequence. This node is one of the PI's of the circuit. It fans out to nodes 12, 13, and 14. The seventeenth position is reserved for the node value which is to be realized by 32-bit data. For example, if the content of the seventeenth position is \([00000, \ldots, 00001, 00]\) (it is a 32-bit word where the first bit from the right denotes the value of the fault-free circuit), then the seventh PI takes 0 as the input value. The third bit from the right indicates that this is the second (i.e., \(3-1=2\); where one bit is reserved for a fault-free circuit) s-a fault in this fault word that causes the input to have a value 1. Furthermore, the second s-a fault is the node 7 s-a-1.

Fault Collapsing and Classification

Individual Node

Consider an AND gate in Figure 3 with all possible combinations of inputs and output. It is shown that if the input s-a-0 (s-a-1) is detectable, then the output s-a-0 (s-a-1) is also detectable. Therefore, there is no need to put both in the fault list, and the number of passes to
simulate the faults can then be further reduced.

\[
\begin{array}{ccc}
A & B & C \\
0 & 0 & 0 & \text{C S-A-1} \\
0 & 1 & 0 & \text{C S-A-1, A S-A-1} \\
1 & 0 & 0 & \text{C S-A-1, B S-A-1} \\
1 & 1 & 1 & \text{C S-A-0, A S-A-0, C S-A-0}
\end{array}
\]

Figure 3. I/O Fault Classification for an AND Gate.

If an input of the node can proceed to the output stage through other paths, then the fault classification property will no longer exist for that input.

**Primary Input Stage**

If at the PI stage, the fan-out nodes are exactly the same among some PI's, then only one sample of those PI's needs to be simulated because the PI's are identical in circuit configuration. One typical example of identical inputs is found in an adder circuit where both inputs go to the same nodes (i.e., XOR & AND gates). This fault collapsing method can further reduce the number of the fault lists and the number of passes.
Testing Decisions

For a traditional parallel simulator (e.g., SCIRTSS), simulation results of the whole fault lists are evaluated for each input test vector. The decision whether the simulation is completed is based on the total fault coverage achieved. However, some fault words may reflect stronger fault coverage that simulation of these fault words can be completed in just a few passes.

MODPAR employs the analysis of separate fault words. By separate treatments, targeted fault coverage for some fault words can be achieved without unnecessary passes. After a fixed number of input test vectors are applied to the circuit, the simulation of each fault word is considered completed if its detection ratio exceeds the target. Otherwise, another testing technique will be used to simulate the faults for that fault word. The algorithm for MODPAR is given in Figure 4. The FORTRAN versions of MODPAR and CONVERT are included in Appendix A.

The target ratio of fault coverage for this study is set at 93%, and each fault word is treated independently. Comparisons of CPU time and fault coverage ratios between SCIRTSS and MODPAR are provided for a few combinational circuits.
Get Circuit Information from CONVERT
Determine the optimum number of input test vectors
Apply fault classification and collapsing techniques
Divide faults into separate fault words
FOR each fault word
  FOR each input test vector
    FOR each node in sequence
      1. Determine the value of node inputs considering fault injection
      2. Perform logic operation
      3. Determine the value of node output considering fault injection
    ENDFOR
  Get the simulation data at the PO's
  IF Total fault coverage exceeds target ratio
    THEN Simulate another fault word
  ENDFOR
  IF Total fault coverage is under minimum requirement
    THEN Simulate the fault word using more specialized techniques*
  ENDFOR
ENDFOR
Re-link the detected faults and the collapsed faults
OUTPUT
END;

Figure 4. Algorithm for MODPAR.

* This technique is the subject of future improvement.
CHAPTER IV

SIMULATION RESULTS

Tested Circuits

Four combinational circuits have been tested.

1. ADD.HPS: A 4-bit ripple carry adder with 8 primary inputs and 5 primary outputs. A total of 186 faults are to be tested for this 34-gates circuit.

2. DCD.HPS: A 3 to 8 decoder circuit with 3 primary inputs, 8 primary outputs, and 19 gates. 103 faults are to be tested.

3. INC.HPS: An incrementor circuit with 4 primary inputs, 4 primary outputs, 10 gates and 50 faults.

4. KMB.HPS: A constructed combinational circuit; 4 primary inputs, 2 primary outputs, 24 gates, and 130 faults.

Circuit description and the number of faults are included in Table 5. The HPS (Hardware Programming Sequence) files for these four circuits are included in Appendix B.

Summary of Results

Data obtained from MODPAR and SCIRTSS (Hill & Huey,
are included in Table 6. From this table, it is obvious that MODPAR enjoys advantages over SCIRTSS on the simulation time for these four combinational circuits.

Table 5
Circuit Description and Numbers of Total and Collapsed Faults for Benchmark Circuits

<table>
<thead>
<tr>
<th>Tested Circuit</th>
<th>PI's</th>
<th>PO's</th>
<th>Gates</th>
<th>Faults</th>
<th>Collapsed Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.HPS</td>
<td>8</td>
<td>5</td>
<td>34</td>
<td>186</td>
<td>36</td>
</tr>
<tr>
<td>INC.HPS</td>
<td>4</td>
<td>4</td>
<td>10</td>
<td>50</td>
<td>10</td>
</tr>
<tr>
<td>KMB.HPS</td>
<td>4</td>
<td>2</td>
<td>24</td>
<td>130</td>
<td>34</td>
</tr>
<tr>
<td>DCD.HPS</td>
<td>3</td>
<td>8</td>
<td>19</td>
<td>103</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 6
Simulation Data for SCIRTSS & MODPAR

<table>
<thead>
<tr>
<th>Tested Circuit</th>
<th>SCIRTSS</th>
<th>MODPAR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fd.</td>
<td>Fc.</td>
</tr>
<tr>
<td>ADD.ST3</td>
<td>73</td>
<td>85%</td>
</tr>
<tr>
<td>INC.ST3</td>
<td>41</td>
<td>100%</td>
</tr>
<tr>
<td>KMB.ST3</td>
<td>113</td>
<td>95%</td>
</tr>
<tr>
<td>DCD.ST3</td>
<td>84</td>
<td>100%</td>
</tr>
</tbody>
</table>

Fd: Faults detected
Fc: Fault coverage
CHAPTER V

FUTURE IMPROVEMENT

As illustrated in Chapter IV, MODPAR provides an efficient tool for fault analysis of combinational circuits. However, it is desirable to extend it to handle sophisticated sequential circuits. MODPAR assumes no Flip-Flop in the circuit, and therefore no clocked storage element is required. When it comes to sequential circuits, clocked simulation is necessary and some node values must be updated to handle feedback signals.

A method called LSSD (level sensitive scan design) (Komonytsky, 1982) provides the linking between sequential circuit testing and combinational circuit testing. By partitioning sequential circuits, MODPAR can be used, while imposing input values at the pseudo inputs, to simulate combinational subcircuits. However, LSSD is employed at the price of a small overhead in circuit complexity.

Even-driven techniques may also be incorporated in MODPAR to enhance the performance. An additional checking program is needed to determine whether an activity has occurred. A significant reduction in computation can be achieved by simulating only active elements.
Appendix A

FORTRAN Version of Programs "MODPAR" and "CONVERT"
This program is a modified parallel fault simulator (MODPAR), which is written as part of the thesis program!

Shih-Ming Li April, 1991

protocols: input: 'Data.dat' specifies the I/O connection of the tested circuit which is the output file of the program 'li.for'.

output: value of the input vectors, s-a faults detected and also the CPU time used.

** declaration

parameter (imaxin=6)
real ti
integer*4 ix(500,6),ir(300,17),result2(1000),ieq(6,6)
integer*4 iy(500),idis(500,20)
integer*4 listnum,invector(imaxin),cr(500,18)
integer*4 iovector,result(50,32),b(33),a,isum(100)

open 'convert' output file: the.dat

OPEN (21, FILE='RAND.DAT', STATUS='UNKNOWN')
open (20, file='the.dat', status='unknown')
read(20,*) inodnum
DO J=1,10
    READ(21,111) (IR(J,I), I=1,10)
ENDDO

begin cpu timer

TI=SECNDS(0.0)

itotft=0
icount1=0
icount=0
iclass1=0
iclass2=0

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
c determine the total faults

do inode=1,inodnum
   read(20,249) (cr(inode,j), j=1,16)
   itotft=itotft+1+cr(inode,3)
   if (cr(inode,2) .eq. 0) then
      icount=icount+1
      itotft=itotft-1
   elseif(cr(inode,2) .eq. 1) then
      icount1=icount1+1
      itotft=itotft-2
   endif
endo

c determine the fault collapsing inputs

do i=1,icount
   do j=i+1,icount
      itk=0
      519 if (cr(i,10+itk) .eq. cr(j,10+itk)) then
      if (itk .eq. cr(i,10)) then
         iclass1=iclass1+1
         goto 520
      endif
      itk=itk+1
      goto 519
      endif
      520 if (itk .eq. cr(i,10)) then
         ieq(i,j)=1
         ieq(j,i)=1
      endif
endo
endo

c determine the fault classification gate

do inode=icount+1,inodnum-icount1
   do j=4,3+cr(inode,3)
      if( cr(cr(inode,j),10) .eq.1) then
         iclass2=iclass2+1
         ix(inode,j-3)=1
      endif
endo
   521 c=0
endo

c initialization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
c get the number of collapsed faults
c
doi pass=1, listnum
do i=1, 32
    result(ipass, i) = 0
enddo
doi do
totft = totft * 2
    totft1 = totft - 2 * (iclass1 + iclass2)
CCC write(*, 250) icount
CCC write(*, 248) icountl
CCC write(*, 251) totft
CCC write(*, 241) iclass1
CCC write(*, 242) iclass2
CCC write(*, 243) totft1
241 format(//, ' There are ', i4, ' pair(s) of inputs to be identical !')
242 format(//, ',i4,' node-output faults are to be discarded. ')
243 format(//, ' Totally', i5, ' classified faults are analyzed ! ')
248 format(//, ' There are ', i2, ' primary output in this circuit, ')
249 format(1x, 16i4)
250 format(//, ',i2,' primary inputs in this circuit, ')
251 format(//, ' and ', i5, ' s-a fault to be tested. ')
252 format(//, ' They are divided into ', i4, ' group : ')
c
c
c
c
c
c
c
determine the number of test vector & fault words
c
totft - int((totft - 0.5) / 30) + 1
ITMAX = MIN(10, 2**ICOUNT)

111 FORMAT(10I3)
CCC write(*, 252) listnum
c
c
c
begin simulation:
c for each fault word:
c
do ipass = 1, listnum
c
c
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
ccc write(*,254) ipass

ccc for each test vector:

do itest=1, itmax
    is=0
ccc write(*,253) itest
253 format(//,' ***** for # ',i2,' test vector')
254 format(//,' ********* for # ',i4,' fault words')
    ifault=0

ccc begin the simulation sequence

do inode=1, inodnum

igtype=cr(inode,2)
ifinnum=cr(inode,3)
ifoutnum=cr(inode,10)

ccc impose random input

if (igtype .eq.0) then
    ir(itest,inode)=nint(ran((itest*29198)+(inode*31)**2))
    if(ir(itest,inode).eq.1) then
        cr(inode,17)=-1
    elseif(ir(itest,inode).eq.0) then
        cr(inode,17)=0
    endif

ccc write(*,270) inode,ir
goto 901
elseif(igtype .eq. 1) then
    cr(inode,17)=cr(cr(inode,4),17)
goto 901
endif

if(igtype .ne.0 .and. is.eq.0) then
    is=1
ccc write(*,271) (ir(itest,k),k=1,icount)
endif
ccc271 format(/,' the input vector is',15i3)

end

c input fault injection
c
do i=1,ifinnum
invector(i)=cr(cr(inode,i+3),17)
if (ifinnum .eq. 2 .and. i.eq. 2) then
if (ieq(cr(inode,4),cr(inode,5)).eq.1) then
  iy(inode)=1
  goto 272
endif
endif
ifault=ifault+2
if (ifault .gt. 30*(ipass-1) .and. ifault .le. 1 30*ipass) then
  ift=mod(ifault,30)
  if(ift .eq. 0) then
    ift=30
  endif
  call mvbits(2,0,2,invector(i),ift-1)
endif
272 c=0
enddo

c c define the type of operation
c c
goto (702,702,703,704,705,706) igtype
c c c

##### and operation #####

702 iovector=-1
do i=1,ifinnum
  iovector=jand(invector(i),iovector)
enddo
goto 900

c ##### nand operation #####

703 iovector=-1
do i=1,ifinnum
  iovector=jand(invector(i),iovector)
end
iovector=jnot(iovect
goto 900

or operation

704
iovector=0
do i=1,ifinnum
   iovector=jior(invector(i),iovector)
enddo
goto 900

nor operation

705
iovector=0
do i=1,ifinnum
   iovector=jior(invector(i),iovector)
enddo
iovector=jnot(iovector)
goto 900

xor operation

706
iovector=0
do i=1,ifinnum
   iovector=jior(invector(i),iovector)
enddo

goto 900

final output

900
cr(inode,17)=iovector
do ite=1,cr(inode,3)
   if(ix(inode,ite) .eq. 1) then
      goto 901
   endif
enddo

output fault injection

ifault=ifault+2
if(ifault .le. 30*ipass .and. ifault
1 .gt. (ipass-1)*30) then
   ift=mod(ifault,30)
   if (ift .eq. 0) then
      ift=30
   endif
   call mvbits(2,0,2,cr(inode,17),ift-1)
endif
901 c=0
enddo
c
c get simulation data from the primary outputs
c
do k=1,icount1
call words(cr(inodnum+k-icountl,17),b)
do i=2,31
   if (b(i) .ne. b(1)) then
      cwrite(*,261) i-1
   endif
enddo
itt=0
do i=2,31
   if(result(ipass,i).eq.1 )  then
      itt=itt+1
   endif
enddo
if (itest .eq. itmax) then
   if (itt .le. 10) then
      ccccc write(*,255) ipass,itt
      256 format(//,' Fault word #',i3,' is completed !
1 and',f8.4,' % of faults has been detected in this
word! ')
      255 format(//,' Fault word #',i3,' is discarded since
1 only',i2,' faults are detected!')
goto 34
   elseif (itt .ge. 28) then
      cccc write(*,256) ipass,itt*100./30.
goto 34
   endif
endif
259 format(//,' so far',i5,' faults have been detected
!',/)
260 format(' the # ',i2,' input vector is:')
261 format(' fault # ',i4,' is detected!')
endo
d
33 isum(ipass)=0
c=0

enddo

262 format(/, 'Totally, ',i2, ' faults are detected in this word. ')

c
c
c cpu time halt!

c
ti=secnds(ti)

c

output stage

**********

ism=0
do ipass=1,listnum
isum(ipass)=0
write(*,269)
269 format( '=================================')
cccc write(*,271) ipass
271 format(/, 'Fault word #',i3,/) 
do i=2,31
 result2(30*(ipass-1)+i-1)=result(ipass,i)
isum(ipass)=isum(ipass)+result(ipass,i)
cccc write(*,270) 30*(ipass-1)+i-1,result(ipass,i)
270 format('Value of fault #',i4,' is',i2)
enddo
cccc write(*,268) isum(ipass)*100./30.
268 format(/,'Fault coverage of this fault word:',f8.4,')

write(*,269)
ism=ism+isum(ipass)
enddo

ih=0
do inode=icount+1,inodnum-icount1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
iq=0
820 ih=ih+2
iq=iq+2
idis(inode,iq-1)=result2(ih-1)
idis(inode,iq)=result2(ih)
if (iy(inode).eq.1) then
iq=iq+2
idis(inode,iq-1)=idis(inode,1)
idis(inode,iq)=idis(inode,2)
endif
if (iq/2 .eq.cr(inode,3)) then
goto 821
endif
goto 820
821 do ite=1,cr(inode,3)
if (ix(inode,ite) .eq.1) then
iq=iq+2
idis(inode,iq-1)=idis(inode,2*ite-1)
idis(inode,iq)=idis(inode,2*ite)
goto 822
endif
enddo
ih=ih+2
iq=iq+2
idis(inode,iq-1)=result2(ih-1)
idis(inode,iq)=result2(ih)
write(*,*)
enddo
isml=0
do inode=1+icount,inodnum-icount1
do j=1,2*cr(inode,3)+2
if (idis(inode,j) .eq.1) then
write(*,109) inode,cr(inode,2),j
isml=isml+1
endif
enddo
enndo
109 format(' node #',i3, '; gtype:',i2,' fault#',i2,' is detected')
write(*,106) isml
write(*,105) isml*100./itotft
106 format(' Totally ',i5,' faults have been detected! ')
105 format(' The fault coverage is',f8.4, '%')
write(*,104) ti
104 format(' The total CPU time used is ',f14.5,3x,'seconds',///)
close (20)
stop
subroutine for bit data realization

subroutine words (a,b)
integer a,b(33)
b(33)=0
do i=1,31
   b(i)=0
   call mvbits(a,i-1,1,b(i),0)
   b(33)=b(33)+b(i)
endo
return
end
Program name: Convert.for

Protocol: input: stage 3 output

output: the.dat; which is a new data format of circuit description.

MODPAR will invoke the.dat as input

IMPLICIT INTEGER(A-Z)

C O M M O N /CDD/NEL,NFF,NCS,NEX,GCD(10000,7),OCOUNT,GCOUNT,NGFO
C O M M O N / X 1 2 /
INSYM(6,200),OUTSYM(6,200),MEMSYM(6,1000),IP,OP,MP
COMMON /X20/ MODNUM(1000),CSP,WClOES3
COMMON /ERFLGS/ST1ERR,COMER

open(23,file='the.dat',status='unknown')
integer matr(10000,16),bass(10000,16),ind1(6),ind2(6),ic(6)
real TI
real DATA IP,OP,MP,CSP/0,0,0,0/
open(23,file='the.dat',status='unknown')
WC10E3=1000
TYPE*,''

CALL OPFIL1('ENTER FILE NAME OF STAGE3
OUTPUT: ',INFTMP,0,1,
1 FILE)

CPU time begin to count

ti=secnds(0.)
READ(INFTMP,*) N
READ(INFTMP,32) NEL,NFF,NCS,NEX,OCOUNT,GCOUNT
32 FORMAT(1X,6I10)
I=0
50 I=I+1
READ(INFTMP,1000) (TEMTAB(J), J=1,6)
1000 FORMAT(1X,2A4,2X,I10,I10,I10,I10)
IF(TEMTAB(1).EQ.'CSFF') THEN
CSP=CSP+1
MODNUM(CSP) = TEMTAB(6) * WC10E3 + TEMTAB(5)
ENDIF
 IF(TEMTAB(3). LE. NCS) GO TO 95
 IF(TEMTAB(3). LE. 11000) THEN
   MP = MP + 1
 DO 70 J = 1, 6
 70 MEMSYM(J, MP) = TEMTAB(J)
 ELSE IF(TEMTAB(3). LE. 11200) THEN
   IP = IP + 1
 DO 80 J = 1, 6
 80 INSYM(J, IP) = TEMTAB(J)
 ELSE
   OP = OP + 1
 DO 90 J = 1, 6
 90 OUTSYM(J, OP) = TEMTAB(J)
ENDIF
95 IF(I .LT. N) GO TO 50
 NGFO = GCOUNT + NFF + OCOUNT

C____________________________________________________________________________
C
C get circuit description data
C
READ(INFTMP, 31) ((GCD(I, J), J = 1, 7), I = 1, NGFO)
C
  it = 0
  ignum = ngfo
 DO 100 J = 1, 7
 100 GCD(NGFO + 1, J) = 0
 DO I = 1, NGFO
   DO J = 2, 7
     IF(GCD(I, J) . LE. NFF) THEN
       GCD(I, J) = 0
     ENDIF
   ENDDO
 ENDDO
31 FORMAT(1X, A4, 5X, 6I8)

C
C gate realization
C
DO I = 1, NGFO
  IF (GCD(I, 1) . EQ. ' DFF ') THEN
    ignum = ignum - 1
    it = it + 1
    GOTO 801
  ELSEIF (GCD(I, 1) . EQ. ' AND ') THEN
    MATR(I, 2) = 2
  ELSEIF (GCD(I, 1) . EQ. ' OR ') THEN
    MATR(I, 2) = 4
  ELSEIF (GCD(I, 1) . EQ. ' NAND ') THEN
    MATR(I, 2) = 3
  ELSE
    ...
elseif (gcd(i, l) .eq. 'NOR ') then
matr(i, 2) = 5
elseif (gcd(i, l) .eq. 'XOR ') then
matr(i, 2) = 6
endif
matr(i, l) = i

get fan-in connection

call ffin(i, ifinnum, ind1, gcd)
matr(i, 3) = ifinnum
c write(*, *) ifinnum, (ind1(j), j=1, 6)
do j=1, ifinnum
matr(i, j+3) = ind1(j)
enddo

get fan-out connection

call ffout(i, ignum, ifoutnum, ind2, gcd, nff)
matr(i, 10) = ifoutnum
do j=1, ifoutnum
matr(i, j+10) = ind2(j)
enddo
write(*, *) (matr(i, j), j=1, 16)

enddo
do i=11001, nex+11000
  call ffout(i, ignum, ifoutnum, ind3, gcd, nff)
bass(i-11000, 1) = i-11000
bass(i-11000, 2) = 0
bass(i-11000, 10) = ifoutnum
do j=1, ifoutnum
  bass(i-11000, j+10) = ind3(j) + nex-nff
enddo
write(*, *) (bass(i-11000, j), j=1, 16)
write(*, *) '_________________________'
enddo

do i=nex+1, nex+ignum
  bass(i, 1) = i
  do j=2, 16
    bass(i, j) = matr(i+nff-nex, j)
  enddo
  do k=4, 16

if (k .eq. 10) then
  goto 234
endif
if (bass(i,k) .ne.0.and.abs(bass(i,k)) .le.1e.1000)
  then
    bass(i,k)=bass(i,k)-nff+nex
  elseif (abs(bass(i,k)) .ge.11000) then
    bass(i,k)=abs(bass(i,k)) -11000
  endif
enddo
c write(*,*) (bass(i,k),k=1,16)
c write(*,*) '+++++++++++++++++++++++++'
endo
do i=nex+ignum-ocount+1,nex+ignum
  bass(i,10)=1
  bass(i,11)=i+ocount
endo
do i=nex+ignum+1,nex+ignum+ocount
c  call ffin(i,ib,ic,gcd)
c  bass(i,1)=i
  bass(i,2)=1
  bass(i,3)=1
  bass(i,4)=i-ocount
cenddo
CLOSE(UNIT=INFTMP)
c write(23,*) ignum+nex+ocount
do i=1,ignum+nex+ocount
c c
CCC write(*,11) (bass(i,j),j=1,16)
c write(23,11) (bass(i,j),j=1,16)
endo
11 format(1x,16i4)
TI=SECONDS(TI)
WRITE(*,299) TI
299 FORMAT(//,' THE CPU TIME USED IS',F10.4,' SENONDS.')
c return
c close(23)
c stop
c subroutine to handle fan-in data transformation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
subroutine ffin(ia,ib,ic,gcd)
integer gcd(10000,7)
dimension ic(6)
ib=0
do k=1,6
   ic(k)=0
enddo
do k=2,7
   if (gcd(ia,k) .ne. 0) then
      ib=ib+1
      ic(ib)=gcd(ia,k)
   endif
enddo
return
end

subroutine to handle the fan-out data transformation

subroutine ffout(im,ignum,in,ip,gcd,nff)
dimension ip(6)
integer gcd(10000,7)
integer nff
in=0
do k=1,6
   ip(k)=0
enddo
do j=nff,ignum+nff+nff
   do k=2,7
      if(gcd(j,k) .eq. im) then
         in=in+1
         ip(in)=j
      endif
   enddo
enddo
return
end

C****************************************************
SUBROUTINE OPFIL1(MESAGE,FUNIT,MOD,FMT,FILE)
INTEGER FUNIT,MOD,FMT
INTEGER STP
CHARACTER*50 FILE,BLANK,MESAGE*36
CHARACTER*3 STA
CHARACTER*11 FMTS

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
DATA BLANK/
1 IF(MOD.EQ.0) THEN
STA='OLD'
ELSE
STA='NEW'
END IF
IF(FMT.EQ.0) THEN
FMTS='UNFORMATTED'
ELSE
FMTS='FORMATTED'
END IF
WRITE(6,200) MESSAGE
FILE=BLANK
READ(6,300) FILE
5 FUNIT=1
OPEN(UNIT=FUNIT,FILE=FILE,STATUS=STA,FORM=FMTS,ERR=10)
200 FORMAT(1X,A36,$)
300 FORMAT(A50)
RETURN
10 TYPE *,' FILE OPENING ERROR ... SIMULATION ABORTED'
STOP
END
Appendix B

HPS Files for Tested Circuits
ADD.HPS

MODULE: PROCESSOR.
EXINPUTS: CLOCK; IN1[4]; IN2[4].
EXOUTPUTS: OUT2[5].

BIDY SEQUENCE: CLOCK.
1 OUT2 = ADD(IN1; IN2); => (1).
ENDSEQUENCE
CONTROLRESET(1).
END.

CLU: ADDER(OP1; OP2) (I)
INPUTS : OP1[I]; OP2[I].
OUTPUTS : S[I+1].
CTERMS : C[I+1].
BODY
  C[I] = 0;
  FOR M = I-1 TO 0 CONSTRUCTAT
    C[M], S[M+1] = BITADDER(C[M+1]; OP1[M]; OP2[M])
  ROF;
  S[0] = C[0].
END.

CLU : FULLADDER(CC; B1; B2).
INPUTS : CC; B1; B2.
OUTPUTS : O0[2].
BODY
  O0[0] = CC & B1 & B2;
END.
DCD.HPS

MODULE : PROCESSOR.
EXINPUTS : CLOCK; IN1[3].
EXOUTPUTS: OUT3[8].
CLUNITS : DCD[8] <: DECODER{3}.

BODY SEQUENCE: CLOCK.
1 => (1).
ENDSEQUENCE
CONTROLRESET(1);.
OUT = DCD(IN1).
END.

CLU : DECODER(IN) {I}.
INPUTS : IN[I].
OUTPUTS : OUT[2^I].
CTERMS : RESULT[2^I].
BODY
FOR M=0 TO (2^I)-1 CONSTRUCT
RESULT[M] = &/TERM(M;IN)
ROF;
OUT = RESULT.
END.
INC.HPS

MODULE : PROCESSOR.
EXINPUTS : IN1[4];CLOCK.
EXOUTPUTS: OUT[4].

BODY SEQUENCE:CLOCK.
1 =>(1).
ENDSEQUENCE
CONTROLRESET(1);
OUT1 = INC(IN1).
END.

CLU : INCER(X) {I}.
INPUTS : X[I].
OUTPUTS : Y[I].
CTERMS : RESULT[I];CARRY[I-1].
BODY
CARRY[I-2].RESULT[I-1] = X[I-1].~X[I-1];
FOR J=I-2 TO 0 CONSTRUCT
RESULT[J] = CARRY[J] @ X[J];
IF J<>0 THEN CARRY[J-1]=CARRY[J]&X[J]
FI
ROF;
Y = RESULT.
END.
KMB.HPS

MODULE : NETMOD.
  EXINPUTS : XIN<2>[1]; YBIN[2]; CLOCK.
  EXOUTPUTS : ZOUT<2>[1].
  PULSES : CLOCK.
BODY SEQUENCE : CLOCK.
  1   =>(1).
ENDSEQUENCE
CONTROLRESET(1);
ZOUT=NET(XIN; YBIN).
END.

CLU     : ITERM(X; YB).
INPUTS  : X<2>[1]; YB[2].
OUTPUTS : Z<2>.
CTERMS  : Y<2>[2].
BODY
  Y<0>[0] = YB[1] + X<0> & ~YB[0];
  Y<0>[1] = ~X<0>& YB[0] + X<0>& YB[0] + X<0>& Y<1-
  1>[1];
  Z<0> = X<0>& (YB[0]@YB[1]) + ~X<0>& YB[0];
FOR I=1 TO 1 CONSTRUCT
  Y<I>[0] = Y<I-1>[1] + X<I> & ~Y<I-1>[0];
  Y<I>[1] = ~X<I>& Y<I-1>[0] + X<I>& Y<I-1>[0] +
           X<I>& Y<I-1>[1];
  Z<I> = X<I>& (Y<I-1>[0]@Y<I-1>[1]) + ~X<I>& Y<I-
  1>[0]
END.
ROF.
BIBLIOGRAPHY


