TECHNION - Israel Institute of Technology

Computer Science Department

WHISTLE - A WORKBENCH FOR TEST GENERATION USING A HIGH LEVEL DESIGN DESCRIPTION

by

R. Renous, G.M. Silberman, I. Spillinger

Technical Report #499

February 1988
WHISTLE - A WORKBENCH FOR TEST GENERATION USING A HIGH LEVEL DESIGN DESCRIPTION

R. Renous*, G.M. Silberman* and I. Spillinger**

* Department of Computer Science, TECHNION, Haifa 32000, Israel
** Department of Electrical Engineering, TECHNION, Haifa 32000, Israel

ABSTRACT

We describe WHISTLE, an environment for the development of test patterns for VLSI designs, based on their description by a high-level hardware description language. The tools in WHISTLE are based on the Difference Fault Model (DFM) approach to fault simulation and test generation. These tools include programs for the analysis of blocks in the design library, generation of code for fault simulation, evaluation of the implementation fault coverage for a set of test patterns, and support for the generation of biased-random test patterns.

1 INTRODUCTION

The techniques and software tools used for test generation and fault simulation of logic designs over the past few years have not kept pace with advances in VLSI technology. Problems with efficiency [1], capacity, and sometimes inadequacy of the fault models to the type of failures present in the new technologies [2], have been of concern for some time. Thus, new approaches for these tasks, more suited to the requirements of VLSI technology, have been the subject of recent research (for a survey, see [3]). Of particular interest are those approaches which rely on a hierarchical representation of the design (e.g., [4, 5]), as they agree with current trends for designing VLSI circuits.

Moving test generation and fault simulation away from the gate level, to a higher level of abstraction (as suggested in [6]), has the potential for supplying the answers to concerns about the efficiency, capacity and fault model adequacy for VLSI designs. However, a major problem in making this transition, resides in the character of the results produced by the high-level tool in general, and particularly in how they are translated to the more familiar measurements of fault coverage at the gate level. I.e., the results obtained by performing high-level fault simulation, must be expressed in terms of fault coverage for the devices and signals which are familiar to the designer. Several efforts in this direction have been reported in the literature (for a survey see [7]), but none has produced consistently accurate results.

In the following, we adopt the Difference Fault Model (DFM) approach [5] to the simulation of high-level designs, and build a complete environment for test generation and fault simulation of designs expressed in a Hardware Description Language (HDL), such as VHDL [8]. This environment, called WHISTLE (Workbench for High-level Simulation and Test-development), consists of a set of programs which support a library-based [9] design style. Among the tools provided by WHISTLE, we distinguish three groups.
First, we have tools for the analysis of the blocks in the design library. Second, a high-level fault simulator which produces results which can be mapped to fault coverage at a lower (e.g., gate) level. The third group of tools provides support for high-level automatic test pattern generation, in the form of a list of blocks with low fault coverage and diverse statistics measured during fault simulation. We should clarify at this point that we see a high-level design as the interconnection of functional blocks which belong to a given design library. In this context, “low level” refers to whatever description (e.g., gates, transistors) is used for the blocks in the library. Furthermore, since the DFM approach is independent of the fault model adopted for the low level (e.g., stuck-at, stuck-open, bridges, etc), our tools can support a variety of such models.

WHISTLE runs under IBM's VM/CMS, and was designed to automate as many facets of the test generation and fault simulation tasks as possible. When that is not possible, WHISTLE provides a set of interactive, easy-to-use menu screens, designed for simplicity and user-friendliness. This will be illustrated in the next section, as we present the tools for analyzing the library blocks. Section 3 will deal with the generation of the code for fault simulation, including integrated control logic for driving the simulation and performing fault injection. Analysis of the results obtained during fault simulation, and their use in biased-random test pattern generation [12], constitutes the subject of section 4. Section 5 shows the accuracy of WHISTLE, illustrated by comparison to direct fault simulation at the gate level, on a number of combinatorial designs.

2 LIBRARY ANALYSIS

In this section we describe the tools in WHISTLE which are used in the analysis of each block in the design library. From this analysis we obtain the functional fault model. This model consists of three main pieces of information, namely (a) the faults to be used during fault simulation of the design, (b) under what conditions each such fault manifests itself, and (c) what measurements should be made during the course of the simulation process. Figure 1 shows the different programs of WHISTLE and their relations. As mentioned above, we derive the functional fault model according to the DFM approach [5].
Let us now consider a single block in the design library, and examine the effect of implementation faults (I-faults) on its behavior. Assume that for some input vector \( v \), the output from this block is the set of \( n \) bits \( b_0, b_1, \ldots, b_{n-1} \), representing the number \( g = b_0 \times 2^0 + b_1 \times 2^1 + \cdots + b_{n-1} \times 2^{n-1} \). The effect of an I-fault on the number \( g \), if present for input \( v \), is to cause the output to be \( w \) instead of \( g \). We can express the functional effect of this I-fault under input \( v \), as the arithmetic difference \( d = w - g \). This difference \( d \) may be common to other pairings of I-faults and input vectors. Therefore, we assign functional faults (F-faults) to a given block to match all the values that \( d \) can attain, and thus the set of all possible F-faults (the fault model) associated with a block is given by \( FF = \bigcup_{v \& i} d_{ij} \), where \( d_{ij} \) is the difference associated with the pair "input vector \( v \) and I-fault \( i \)."

**Example 1**

To illustrate the process for determining the DFM for a functional block type, let us consider the decoder shown in Figure 2. If we assume a single-stuck-at implementation fault model, the list of I-faults for this block includes 24 faults, as listed in Table 1. Table 2 then shows the differences \( d_{ij} \) obtained by simulating all I-faults, using all (four) possible input combinations. By examining this table (we refer to them as the Difference or D-matrix), we see that the set of all possible F-faults in this case is given by \( FF = \{ \pm 1, \pm 2, \pm 3, \pm 4, \pm 6, \pm 8 \} \).
Figure 2. A 2-to-4 Decoder.

Table 1. List of I-faults for the Decoder in Figure 2.

<table>
<thead>
<tr>
<th>I-fault Number</th>
<th>Fault Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>right input to gate 3 stuck at 1.</td>
</tr>
<tr>
<td>2</td>
<td>left input to gate 3 stuck at 1.</td>
</tr>
<tr>
<td>3</td>
<td>right input to gate 4 stuck at 1.</td>
</tr>
<tr>
<td>4</td>
<td>left input to gate 4 stuck at 1.</td>
</tr>
<tr>
<td>5</td>
<td>right input to gate 5 stuck at 1.</td>
</tr>
<tr>
<td>6</td>
<td>left input to gate 5 stuck at 1.</td>
</tr>
<tr>
<td>7</td>
<td>right input to gate 6 stuck at 1.</td>
</tr>
<tr>
<td>8</td>
<td>left input to gate 6 stuck at 1.</td>
</tr>
<tr>
<td>9</td>
<td>input line C₀ stuck at 0.</td>
</tr>
<tr>
<td>10</td>
<td>input line C₀ stuck at 1.</td>
</tr>
<tr>
<td>11</td>
<td>input line C₁ stuck at 0.</td>
</tr>
<tr>
<td>12</td>
<td>input line C₁ stuck at 1.</td>
</tr>
<tr>
<td>13</td>
<td>output line of gate 1 stuck at 0.</td>
</tr>
<tr>
<td>14</td>
<td>output line of gate 1 stuck at 1.</td>
</tr>
<tr>
<td>15</td>
<td>output line of gate 2 stuck at 0.</td>
</tr>
<tr>
<td>16</td>
<td>output line of gate 2 stuck at 1.</td>
</tr>
<tr>
<td>17</td>
<td>output line of gate 3 stuck at 0.</td>
</tr>
<tr>
<td>18</td>
<td>output line of gate 3 stuck at 1.</td>
</tr>
<tr>
<td>19</td>
<td>output line of gate 4 stuck at 0.</td>
</tr>
<tr>
<td>20</td>
<td>output line of gate 4 stuck at 1.</td>
</tr>
<tr>
<td>21</td>
<td>output line of gate 5 stuck at 0.</td>
</tr>
<tr>
<td>22</td>
<td>output line of gate 5 stuck at 1.</td>
</tr>
<tr>
<td>23</td>
<td>output line of gate 6 stuck at 0.</td>
</tr>
<tr>
<td>24</td>
<td>output line of gate 6 stuck at 1.</td>
</tr>
</tbody>
</table>
Table 2. Difference Matrix for the Decoder in Figure 2.

| C_1 | C_2 | g | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 0 0 | 1 0 | 0 2 | 0 4 | 0 0 | 0 1 | 0 0 | 3 0 | -1 0 | -1 | 0 | 0 | 2 | 0 | 4 | 0 | 8 | 0 | 2 | 0 | 4 | 0 | 8 |
| 0 1 | 2 1 | 0 0 | 0 0 | 0 8 | -1 0 | 0 6 | 0 1 | -2 0 | 0 | 1 | -2 | 0 | 0 | 4 | 0 | 8 | 0 | 2 | 0 | 4 | 0 | 8 |
| 1 0 | 4 0 | 1 0 | 0 0 | 0 8 | 0 0 | 4 0 | -3 0 | -4 0 | 0 | 1 | 0 | 1 | 0 | 2 | -4 | 0 | 0 | 8 | 0 | 2 | 0 | 4 | 0 | 8 |
| 1 1 | 8 0 | 0 2 | 4 0 | 0 0 | 4 0 | 0 0 | -4 0 | -6 0 | 0 | 4 | 0 | 2 | 0 | 1 | 0 | 2 | 0 | 4 | -8 | 0 |

Block analysis begins with the program DMAT (refer to Figure 1), which takes as input the implementation of a given block (in our prototype we use a gate-level representation), and the given implementation fault model (we use single-stuck-at), and builds the D-matrix for the block. The F-faults associated with this block are then extracted from the D-matrix in a straightforward fashion.

Let us now turn our attention to the other two components of the functional fault model, namely the data for the fault injection process, and what measurements should be performed during simulation. In principle, the information we need is contained in the D-matrix (as produced by DMAT) for each block, but keeping the full matrix for large blocks might be impractical in terms of both storage and performance. Therefore, we look for ways to extract and compress the relevant information from the D-matrix, without compromising the accuracy of the results obtained (for further details, refer to [10]). In WHISTLE, the user is given the power to make the trade-off between an increase in accuracy and the associated space/time penalties, during the analysis of each library block.

The data for fault injection and fault coverage calculations, as produced by DMAT for each block in the library, consists of the following tables:

1. For every output bit $s$ of the block, two tables are created
   a. $V_o^s$ - the off-set of $s$, a list of all input vectors for which bit $s$ of the (fault-free) output has the value 0; and
   b. $V_i^s$ - the on-set of $s$, the list of all input vectors for which the bit $s$ of the (fault-free) output has the value 1.

2. For each $ffv \in FF$ of the block
   a. for each I-fault $j$, $SF_{ffv, j}$, a list of the inputs to the block which "excite" this I-fault and causes the appearance of $F$-fault $ffv$ at the block's output.
   b. MBD$_{ffv}$, a list of inputs which cause a multiple bit distortion on the block's output.
   c. SBD$_{ffv}$, a list of inputs which cause a single bit distortion on the block's output, only for the case where $ffv$ is of the form $\pm 2^i$.

For example, for the Decoder of Example 1, $V_o^1 = \{0,1,3\}$, $V_i^1 = \{2\}$, $SBD_s = V_o^1 = \{0,1,3\}$, $MBD_s = \{0\}$, and $SF_{d,2} = \{0,1,3\}$.

The sets (lists) $V_o^s$, $V_i^s$, SBD$_{ffv}$, and MBD$_{ffv}$ are needed for the fault injection process, while the $SF_{ffv, j}$ are used during estimation of the implementation fault coverage (refer to [5, 10]).

Example 2

To further illustrate the above sets, we now introduce a two-bit Full-Adder, similar to FA2 in [11] (see Figure 3). This example will be used throughout the remainder of the paper, to illustrate several features of WHISTLE.
Figure 3. A Two-Bit Full-Adder (FA2)

Running DMAT on FA2 yields the set of functional faults FF = \{-4, -2, -1, 1, 2, 4\}. The sets needed for fault injection, for each \( \mathcal{ff}_\mathcal{f} \in \mathcal{FF} \), are given by

\[
\begin{align*}
\mathcal{MBD}_- &= \emptyset \\
\mathcal{SBD}_- &= \{11, 13, 14, 15, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31\} \\
\mathcal{MBD}_1 &= \{11, 13, 14, 15, 19, 21, 22, 23\} \\
\mathcal{SBD}_1 &= \{3, 5, 6, 7, 8, 9, 10, 12, 16, 17, 18, 20, 27, 29, 30, 31\} \\
\mathcal{MBD}_2 &= \emptyset \\
\mathcal{SBD}_2 &= \{1, 2, 4, 7, 9, 10, 12, 15, 17, 18, 20, 23, 25, 26, 28, 31\} \\
\mathcal{MBD}_3 &= \emptyset \\
\mathcal{SBD}_3 &= \{0, 3, 5, 6, 8, 11, 13, 14, 16, 19, 21, 22, 24, 27, 29, 30\} \\
\mathcal{MBD}_4 &= \{8, 9, 10, 12, 16, 17, 18, 20\} \\
\mathcal{SBD}_4 &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 16, 17, 18, 20\} \\
\mathcal{V}_1 &= \mathcal{SBD}_1 \\
\mathcal{V}_1 &= \mathcal{SBD}_2 \\
\mathcal{V}_1 &= \mathcal{SBD}_3 \\
\mathcal{V}_1 &= \mathcal{SBD}_4
\end{align*}
\]

Notice that for the FA2 case, every \( \mathcal{ff}_\mathcal{f} \) is of the form \( \pm 2^s (0 \leq s \leq 2) \), with only the F-faults \( \mathcal{ff}_\mathcal{f} = \pm 2 \) having each \( \mathcal{MBD}_\mathcal{f} \neq \emptyset \).

As mentioned above, we aim to compress the information contained in the above tables (as derived from the D-matrix), without losing accuracy in the process. The next subsections will show our approach to this problem, building on the work of [10], and implemented by the pair of programs INJENT, for fault injection, and DETENT, for fault coverage evaluation (refer to Figure 1).
2.1 Characterizing Sets of Vectors

In general, there is no automatic procedure for producing a compact characterization of a set of vectors, other than "trial and error." But, as shown in [10], it is possible to evaluate how good a characterization is, without having to perform extensive experiments and/or calculations. Thus, WHISTLE provides an interactive environment through which the user can select a characterization for a set of vectors, and obtain a measure of its "quality" (accuracy). This measure of quality is given by an evaluation function based on entropy measurements [13]. This evaluation function yields a value inversely proportional to the accuracy of the set characterization, and attains its optimum (minimum) value of zero for that characterization which maintains complete accuracy.

The two programs dealing with evaluation functions, namely INJENT (INJection ENTropy) and DETENT (DETection ENTropy), present the same user interface, and differ only in their respective evaluation functions. In this subsection we shall introduce the common interface, while the two following subsections will deal with the evaluation functions for each program separately.

The common user interface is shown in Figure 4. In it we distinguish the top, middle and bottom areas of the screen. In the top area we see the block’s name (which is a parameter) and characteristics from its library description (number of input and output bits), and data supplied by DMAT (number of functional faults). Immediately following this information, there are two fields to be filled by the user (they are highlighted on the screen), i.e., which F-fault(s) the user wants to process, and a limit to the number of counters to be updated (during fault simulation) for this fault (this is relevant for DETENT, while INJENT uses the default, of 2). The middle area of the screen is allocated to the description of the characterization which the user wants to evaluate, while the bottom part shows Program Function (PF) key definitions and provides means to input CMS commands. All messages (errors and other information) produced by the programs, also appear in the bottom area of the screen.

```
*************** ENTROPY OF BLOCK FA2 ***************
Number of Input Bits: 5  110 (1 . . )
Number of Output Bits: 3  001 01 . .
Number of Functional Faults: 6
For which Fault:    (All, List of Numbers)
Number of Counters: 2

1:  
2:  
3:  
4:  
5:  
6:  
7:  
8:  
9:  
10:  
11:  

CMS Command ---->
PF1= Help   PF2- Calculate  PF3= Quit  PF4= Save
```

Figure 4. User Interface for INJENT and DETENT

Characterization of a set of vectors is described by the user in terms of the characterization function, whose arguments are (some of) the input and/or output bits for the block being studied. This function is expressed in PLIENT (Programming Language/Entropy), a very simple "programming language" whose grammar is
trivial, but allows the user to express a wide variety of functions. In PL/ENT, variable names are just one letter long (maximum is 26 variables), and each such variable represents a string of bits. Individual bits in the string can be referred to by the name of the variable and the position of the bit next to it (e.g., G1, T0), with bit 0 being the Most Significant Bit. No explicit concatenation operator exists, but two variables (or bits) written next to one another (e.g., TS2) are implicitly translated to the concatenation of the two variables.

There are two predefined variables, I and O, which represent the input and output bits of the block, respectively, and their lengths are shown in the top portion of the screen (refer to Figure 4). Separation between two statements in a program is done either by a semi-colon (';') or by a new line. The full grammar of PL/ENT is given in the Appendix. Below we see two examples of PL/ENT programs, as they would be entered by the user.

Example 3

Let us consider the two-bit full adder FA2 of Figure 3, which has 5 input bits (I0 I1 ... I4) and 3 output bits (O0 ... O2). The "program" return 1113 xor 0214 evaluates the characterization function 1

Example 4

For the same FA2 block in example 3, a more sophisticated program for a characterization function is given by:

```
var S(2), X
S = I2 + I3 + I4; X = 10 xor I1
if S0X = '01' then return 1
    else return 0
```

When the user terminates to enter his PL/ENT program, depressing the PF-2 key causes its translation into a PL/I function. This function is compiled and linked to a program which evaluates the "quality" of the characterization function, which is then run, and the result displayed to the user. (The meaning of this result, for INJENT and DETENT, is explained in the following subsections.) If the result obtained does not please the user, he can try modifying the characterization function, or using a completely different one, until satisfied. When this is the case, the PF-4 key will cause association of the given characterization function with the library block (and F-fault) being processed.

The PF-1 (Help) key gives the user a list of simple characterization functions, and can guide him in the (initial) choice of a good function. In the following subsections we examine the evaluation functions for INJENT and DETENT.

2.2 Evaluation Function for INJENT

Following the strategy for fault injection proposed in [10], and described briefly in section 3 below, we need for each $f \in FF$, characterization functions for the sets $SBD_{f}$ and $MBD_{f}$. (The sets $V_{f}$ and $V_{i}$, as produced by DMAT, serve as initial suggestions for characterization functions. This case is illustrated in Example 2, for all the $SBD_{f}$ sets.) Let us assume that we want to produce a characterization function for the vectors in the set $VOI$ (Vectors of Interest), which best distinguishes them from among the vectors in the set $APV$ (All Possible Vectors). (Clearly, VOI=APV, and we are looking for a function which yields values for vectors in VOI which are different from those given when the function is applied to vectors in the
Furthermore, we would like the characterization function to have the smallest possible domain, and be as simple as possible (both these factors affect simulation performance).

To measure the quality of a characterization function $f$, INJENT uses the entropy-based [13] evaluation function given by

$$
E_{\text{INJENT}}(f) = - \sum_{i=1}^{k} [P^i \log_2 P^i + (1 - P^i) \log_2(1 - P^i)]
$$

where $k$ is the size of the domain of $f$, $P^i = |\text{VOI}^i| + |\text{APV}^i|$, and $\text{VOI}^i (\text{APV}^i)$ is the (sub) set of vectors from VOI (APV) for which $f = i$. Notice that $E_{\text{INJENT}}(f) = 0$ means that $(\forall i)[P^i \in \{0, 1\}]$, which is the minimum we look for. Among all characterization functions which yield this minimum, the optimal one is that which exhibits the smallest domain, and is the easiest to calculate during simulation. Also, notice that larger values of $E_{\text{INJENT}}(f)$ indicate lesser quality (i.e., decreased accuracy) of the characterization function $f$.

Notice that in principle, the set APV contains all $2^m$ possible input vectors for a block with $m$ input bits. But, it is sometimes easy, and always advantageous, to restrict APV to a smaller set using a very simple check. For example, an F-fault which causes a single-bit distortion with a positive difference (i.e., $+2'$) cannot possibly exist in conjunction with an input vector $I$ which yields a value of 1 for bit $s$ (i.e., $I \in V^i_s$). The advantage in restricting the size of APV is that there is a better chance that a good characterization function will be simpler.

Let us now consider the VOI sets we want to characterize, i.e., SBD$_{fr}$ and MBD$_{fr}$. First, when $ffv = +2'$ ($-2'$) we have that $\text{APV} = V^i_t (V^i_f)$ for $\text{VOI} = \text{SBD}_{fr}$, and $\text{APV} = V^i_f (V^i_t)$ for $\text{VOI} = \text{MBD}_{fr}$. The only other type of VOI we must consider are those MBD$_{fr}$ sets with $ffv \neq \pm 2'$, for which we cannot restrict APV, and thus $|\text{APV}| = 2^m$.

Example 5

Let us consider the FA2 block of Example 2, together with the characterization function given in Example 4. Applying this function to the set MBD$_2$ (for which $\text{APV} = V^i_f$), we have that

$$
\rho^0 = \frac{|\text{VOI}^0|}{|\text{APV}^0|} = \frac{|\phi|}{|\{3, 5, 6, 7, 27, 29, 30, 31\}|} = \frac{0}{8} = 0
$$

and

$$
\rho^1 = \frac{|\{8, 9, 10, 12, 16, 17, 18, 20\}|}{|\{8, 9, 10, 12, 16, 17, 18, 20\}|} = \frac{8}{8} = 1
$$

and thus $E_{\text{INJENT}} = 0$ which means that this characterization function maintains full accuracy.

2.3 Evaluation Function for DETENT

Whereas INJENT addressed the question for which vectors appearing at the input to a block, a certain F-fault must be injected, DETENT must deal with vectors which caused detection of a given F-fault. In this case, we are interested in knowing which I-fault(s) may be covered by such a detection, and thus we want to characterize input vectors in such a way so as to distinguish among I-faults which correspond to the same
F-fault. Thus, the sets to be characterized in this case are the $SF_{f^k,i'}$. Since now we have a number of sets (one for each of the I-faults in question) for each $f^k$, and they all must be handled by the same characterization function, the previous evaluation function cannot be used. Therefore, we define for each $j = 1 \ldots r$ ($r$ is the number of I-faults in the block for which $SF_{f^k,i'} \neq \phi$) the auxiliary functions

$$e_j(f) = - \sum_{i=1}^{k} [P_i^{j} \log_2 P_i^{j} + (1 - P_i^{j}) \log_2 (1 - P_i^{j})]$$

where $k$ is the size of the domain of $f$, and $P^{j} = |SF_{f^k,j}| + |A_{f^k}|$ with $SF_{f^k,j}$ ($A_{f^k}$) being the (sub) set of vectors from $SF_{f^k,j}$ ($SBD_{f^k} \cup MBD_{f^k}$) for which $f = i$.

The evaluation function is then given by

$$E_{DETENT}(f) = \frac{\sqrt{\sum_{j=1}^{r} (\frac{e_j(f)}{r})^2}}{k}$$

Again, it can be seen that the closer $E_{DETENT}(f)$ is to zero, the more accurate the characterization function is (this will be illustrated in section 5 below). Also, since $E_{DETENT}(f)$ is inversely proportional to $k$, we could get a better characterization function by using a larger $k$, but this would require that we keep track of more information (in the form of counters) during the fault simulation process. This trade-off will become clearer in section 4 below, where we present the formulae for estimating the fault coverage.

This concludes the description of the library analysis tools in WHISTLE. The next section describes the generation of code for the simulation of the F-faults in the design.

### 3 CODE GENERATION FOR FAULT SIMULATION

Although the library analysis phase involves a large amount of work, as it must handle each F-fault within every block in the design library, this phase is done just once, at the time the library is created. Clearly, blocks being added to the library or changes done to their low-level implementation and/or fault model, require that the analysis be repeated. But, these actions are few and far between, once a design library becomes stable. On the other hand, the code generation and fault simulation phase (refer to Figure 1) is repeated countless times for all the designs based on the same library. This section describes the tools in WHISTLE which handle these tasks.

The generation of code to represent the design to be simulated relies on syntax-directed translation techniques [10]. This process is implemented by the HIFAST (High-level FAult SimulaTor) compiler, which analyses the HDL description of the design, and produces PL/1 code according to the information produced by the library analysis phase. Actual simulation proceeds after the PL/1 code is compiled and linked to some service routines.

We should point out at this point that the reason we chose PL/1 as the “object code” for HIFAST was for rapid prototyping. The resulting code is highly readable, and its correspondence to the original design is quite apparent. Nevertheless, as we shall see below, we restrict the PL/1 code to follow a style which facili-
tates its replacement by performance-oriented machine language code, for a production version of WHISTLE.

Code generated by HIFAST consists of three parts, the *declaration part* which corresponds to the definition of the facilities (i.e., signals, latches, etc) in the design, the *block execution part* in which each block’s representation includes both its fault-free and faulty behaviors, and the *global control part* for activating the injection of faults during simulation, gathering of the fault injection/detection data, and supervising overall execution of the code. The first part is a straightforward mapping of the facilities declared in the HDL to variables in PL/I, and we will not extend its description here.

The block execution part has the general structure shown below.

```
Block1:
  Good Output Computation
  Goto Block(2)
Block1f:
  Fault Injection Procedure of Block1
Block2:
  Good Output Computation
  Goto Block(3)
Block2f:
  Fault Injection Procedure of Block2
Block3:
  ...
```

We can see that for each block in the design, HIFAST generates a two-part semantic routine, consisting of:

1. computation of the *fault free output* \( g \) from the block in question; and
2. deciding which fault(s) (if any) are to be injected for this block (depending on its current input), and adding the appropriate difference(s) \( d (=ffv) \) to the good output \( g \) (refer to section 2).

Each label variable Block\( (i) \) is set equal to Block\( i \) for the fault-free execution of the design, which thus proceeds in an *oblivious* fashion. In this manner, the control flow of the original HDL description is maintained. When exercising the code with faults, the global control logic changes the values in the label variables (e.g., Block(2) is set to Block1f when faults are injected into Block1). Also, the overall control flow assumes a more *event-driven* flavor during fault simulation, for reasons of efficiency.

**Example 6**

Let us assume that the following statements appear in a design (we use VHDL [8] as an example only).
component FA2 is
  port (CIN,X0,Y0,X1,Y1 : in BIT;
       Z0,Z1,COUT : out BIT);
end component;
signal A0,A1,B0,B1,C0,S0,S1,C1 : BIT;

F1 : FA2 port map (CIN => C0, X0 => A0, Y0 => B0, X1 => A1, Y1 => B1, Z0 => S0, Z1 => S1, COUT => C1);

The PL/1 code generated by HIFAST in this case is:

DCL (A0,A1,B0,B1,C0,S0,S1,C1) BIT;
%INCLUDE FA2;

BLOCK1:
  RES = FA2(C0,A0,B0,A1,B1); /* fault-free output (g) */
  S0 = SUBSTR(RES,1,1); /* individual output bits */
  S1 = SUBSTR(RES,2,1); C1 = SUBSTR(RES,3,1);
  GO TO BLOCK(2); /* overridden by global control */

BLOCK1F:
  GOOD = RES; /* fault behavior */
  INP = CO || A0 || B0 || A1 || B1; /* the input vector */
  TYPE = 1; /* for the FA2 block */
  CALL FAULT-INJECT; /* fault injection procedure */
  S0 = SUBSTR(OUTP,1,1); /* faulty output bits */
  S1 = SUBSTR(OUTP,2,1);
  C1 = SUBSTR(OUTP,3,1);
  GO TO RETURN-TO-CONTROL; /* end of design */

Notice that the code generated is simple and "flat," in the sense that the use of subroutines and do-loops is minimal, and the use of \texttt{goto} is frequent. As mentioned above, this fits our decision to make the code similar to what we would do at the machine-language level (e.g., notice that no parameters are passed to the fault injection procedure, instead we use the global variables GOOD, INP, TYPE and OUTP). A by-product of this style is the freedom given to the PL/1 compiler to optimize the code.

Let us now look at our implementation of the fault injection procedure FAULT-INJECT.

3.1 FAULT-INJECT - The Fault Injection Procedure

FAULT-INJECT is a general procedure for the injection of faults. It performs those tasks which are common to all types of blocks, and then calls type-specific procedures which handle each block type, according to the characteristics determined by INJENT. FAULT-INJECT uses three global variables as input "parameters," \textit{GOOD}- the good output (g) from the block, \textit{INP}- the current input to the block, and \textit{TYPE}- the block's type. It returns in the global variable \textit{OUTP} the faulty output (w) according to the fault model of that block. Notice that for some values of the input, several faults may need to be injected. In this case an array is used to hold \textit{all} these faults, and successive calls to FAULT-INJECT will produce the faults...
one after the other. Also, FAULT-INJECT saves the state of the design variables so that the fault-free simulation does not have to be repeated in between successive faults.

Figure 5 below shows a (pseudo-code) general template for building a type-specific procedure, according to the information supplied by INJENT. Recall that the objective is to decide which \( ffv(s) \) must be added to the good output \( g \), for the present input to the block.

\[
\text{type-fault-injection-procedure (INP,GOOD) returns: set-of-fault-values}
\]

\[
\begin{align*}
\text{set-of-fault-values} & = \phi \\
\text{for each } ffv \text{ possible} & \\
\text{if } ffv \text{ is of the form } +2^r (-2^r) & \text{ then do} \\
\text{if } SBD_{ffv} = V_b^\ominus (V_b) & \text{ then} \\
\text{if } \text{bit } s \text{ of GOOD} = 0 (1) & \text{ then } x = \text{GOOD with bit } s \text{ inverted} \\
\text{add } x \text{ to set-of-fault-values} & \\
\text{else if } \text{INP } \in SBD_{ffv} & \text{ then } x = \text{GOOD } + ffv \\
\text{add } x \text{ to set-of-fault-values} & \\
\text{else if } \text{INP } \in MBD_{ffv} & \text{ then } x = \text{GOOD } + ffv \\
\text{add } x \text{ to set-of-fault-values} & \\
\text{else if } \text{INP } \in MBD_{ffv} & \text{ then } x = \text{GOOD } + ffv \\
\text{add } x \text{ to set-of-fault-values} &
\end{align*}
\]

Notice the distinction being made between the sets \( SBD_{ffv} \) and \( MBD_{ffv} \). This is done since in a large number of cases where \( ffv = +2^r (-2^r) \), we have that \( SBD_{ffv} = V_b^\ominus \), where \( b \in \{0,1\} \). (Recall that \( V_b^\ominus \) denotes the set of inputs for which the fault-free output bit \( s \) has the value \( b \).) In this particular case, the value of bit \( s \) signals whether \( ffv \) must be injected, and if so, bit \( s \) is simply inverted. Furthermore, when the pair of \( F \)-faults \( \pm 2^r \) fit the above characteristics (e.g., in the case of simple Boolean gates), we can do away with the checking of bit \( s \) and immediately invert it.

In the general case, we must test for inclusion of the input \( \text{INP} \) in one (or both) of the sets \( SBD_{ffv} \) and \( MBD_{ffv} \). This is done by computing the characterization function defined using \( \text{INJENT} \), with \( \text{INP} \) and \( \text{GOOD} \) supplying the arguments.

**Example 7**

The fault injection procedure for the FA2 block is given by:
FA2-fault-injection-procedure (INP,GOOD) returns: set-of-fault-values

del S(2) bit, X(1) bit

set-of-fault-values = ∅

F = GOOD with bit 2 inverted
add F to set-of-fault-values /* Take care of * /
/* ffv = -4 and ffv = 4 * /

F = GOOD with bit 1 inverted
add F to set-of-fault-values /* Take care of * /
/* ffv = -2 and ffv = 2 * /

F = GOOD with bit 0 inverted
add F to set-of-fault-values /* Take care of * /
/* ffv = -1 and ffv = 1 * /

S = INP(2) + INP(3) + INP(4)
X = INP(0) xor INP(1)

if S(0) || X = '11'B then F = GOOD - 2 /* if INP ∈ MBD-2 */
add F to set-of-fault-values

if S(0) || X = '01'B then F = GOOD + 2 /* if INP ∈ MBD+2 */
add F to set-of-fault-values

Notice that we take advantage of those cases (e.g., ffv = ± 4) in which the characterization function is trivial, and perform bit inversions as explained above.

3.2 The Global Control

The logic which controls code execution during fault-free and faulty design simulation has the overall structure shown below.

- While there are input patterns
  - Read the next input pattern
  - Simulate the fault-free machine.
  - For each block \( i \) executed during the fault free simulation
    /* Execute the fault part of the block \( i \) */
    - BLOCK\((i+1)\) = BLOCK/F
    - If number of blocks influenced by block \( i > \text{factor} \) then goto Simulation
    - Order the array BLOCK, s.t. only the relevant blocks will be executed

Simulation:
  - Restore the previous state of the simulator
  - Goto BLOCK(\( i \))
  - If there was detection, update relevant counters
  - If there is more fault machine to run goto Simulation
  - Restore the array BLOCK to its original state

This structure is mostly self-explanatory, but there is one step which necessitates some elaboration. In general, it is possible to assign values to the label variables Block(\( i \)) in such a way so as to bypass those blocks in the faulty design which are not affected by the fault (their outputs retain their fault-free values).
Thus, we keep an influence graph from which this information can be extracted, so that only blocks possibly affected by the fault need to be re-simulated. But, sometimes only a small percentage of the blocks would be skipped, and the re-assignment of values to the labels is a time-consuming operation. Therefore, we empirically determined the trade-off between the time needed to re-assign labels, and the time savings which result from the re-assignment (in the form of blocks which are not re-simulated). This trade-off determines that whenever the block containing the fault being simulated influences more than 20% of the total number of blocks which would be simulated (i.e., appear between the faulty block and the end of the compiled code), re-assignment of the label variables is performed. Otherwise, re-assignment is bypassed, and all blocks “downstream” from the faulty block are re-simulated.

The results from a fault simulation run consists of sets of counters. In the next section we relate these counters to the information produced by DETENT (refer to section 2), and show how they are used to estimate the fault coverage of a design.

4 RESULTS ANALYSIS

Basically, any fault simulation process should yield at least the answer to one question, i.e., which faults were detected by a given set of test patterns. For gate-level fault simulators there is a one-to-one correspondence between the faults as seen by the simulator, and those which are of interest to the designer/user. In our case, this correspondence is slightly more involved, as shown in [10] and summarized below. Within WHISTLE, the tool COVER (refer to Figure 1) handles all tasks related to fault coverage calculations. In addition to this basic mode of operation, in which all F-faults in the design are candidates for detection by the set of test patterns, the fault simulator has two other modes of operation, zoom and profile which provide support for the test pattern generation approach suggested in [12]. Zoom mode is used to examine a particular F-fault in a given block, fault simulate this F-fault for a given set of test patterns, and select from this set those patterns which cause detection of the F-fault. The profile mode is similar to zoom mode, but it fault simulates all the F-faults in the chosen block. Usage of these two modes will become clear when we discuss the test generation support in WHISTLE.

4.1 Estimation of the Implementation Fault Coverage

Let us recall, from Section 2, that for each vector appearing at the input to a block during fault simulation, we apply a characterization function to determine which F-faults must be injected, and in case there was a detection, we want to characterize those I-fault(s) which may be covered by such a detection. From [10], the probability that a given I-fault (in a given block), say I-fault, has been detected, is

$$w_j = 1 - \prod_{\forall ffa \in FF} \prod_{i=1}^{k_{fa}}[(1 - P_i)^{N_{fa}^i}]$$

where \(N_{fa}^i\) counts the number of detections for F-fault \(ffa\), when the characterization function chosen by DETENT for the input to the block containing I-fault, equaled \(i\). (For the definition of \(P_i\) refer to Section 2.3.) Notice that each F-fault may have a different characterization function, and thus their domains may vary in size, a fact reflected by subscripting the domain size \(k\).

From the above formula for \(w_j\) we can clearly see the impact on performance of the characterization functions chosen by DETENT for each \(ffa\). Not only their evaluation should be as efficient as possible, but also
they should have the smallest possible domain $k_{mn}$, since this is the number of counters which must be kept during fault simulation, and thus impacts its memory requirements.

Since the detection probability of each I-fault in a block, as shown above, is independent of other I-faults, the fault coverage of a block is simply given by $W = \sum w_j$. After calculating the fault coverage $W$ for each block in the design, COVER estimates the total fault coverage by dividing their sum by the total number of I-faults in the design.

### 4.2 Support for Test Pattern Generation

The approach suggested in [10] for test pattern generation consists of three phases. Initially, the design is fault-simulated with a set of random patterns, until the fault coverage curve "flattens out." From the fault coverage at this point, obtained using the above formula for the detection probability of each I-fault, those blocks with low fault coverage are selected for processing in the second phase. This phase consists of generating biased-random test patterns, each guaranteed to excite a given F-fault of the block in question, and checking for detection by fault simulation. Also during this phase, for F-faults which are particularly difficult to detect because of observability problems, an attempt is made to obtain a profile for patterns which would "open" paths for observability. The third phase corresponds to deterministic test pattern generation.

By supplying the three modes of operation during fault simulation, WHISTLE supports the above approach. Basic mode is used for the first phase (random test patterns), or for any application which requires evaluating the fault coverage of a set of test patterns. The second phase uses both zoom and profile modes. While the use of zoom is quite straightforward, the profile mode merits some attention. As explained above, this mode picks from a set of vectors all those which cause detection of F-faults in a given block. From this set we must derive some characterization of the patterns which open a path for observability through this block.

As suggested in [10], we produce a set of weights by counting the relative number of 1's in each bit position of the set of patterns, and merge this information with the biasing required for the excitation of an F-fault.

BRTPG consists of two programs, GENTEST for generating a set of biased random test patterns that are guaranteed to excite a given F-fault in the design, and HARDTEST which also incorporates observability weights into the biasing for the test patterns. GENTEST receives from COVER the selected block number and F-fault, and the quantities $N$ and $M$ as defined above. Using a backtracing process similar to the one found in most gate-level test generation algorithms (e.g., D-algorithm [14], PODEM [15], FAN [16]), GENTEST creates a number of templates of 0, 1 or X ("don't care") values for the inputs to the design, which will cause excitation of the chosen F-fault. From these templates, a set of test patterns is produced and evaluated by fault simulation (using the zoom mode), according to the process outlined below.
Call GENTEST
Repeat
    Take a template
    Create a set of $M$ vectors by replacing randomly-every ‘X’ bit by 0 or 1
    Run the simulator to see if there is detection
    Save the vectors for which there was detection
until $N$ detections or no more templates

If no detections are observed, the problem must be with the lack of a path for observability. In this case we invoke the program HARDTEST to attempt a solution. HARDTEST uses the profile mode to obtain information about blocks along possible observability paths, and uses this information to modify the templates produced by GENTEST.

The vectors obtained by applying BRTPG to a given F-fault are run again through the fault simulator, this time in basic mode, to evaluate their true contribution to the overall fault coverage of the design. (This is done in an incremental fashion [17] for performance reasons.) This whole process is repeated until it is estimated that the effort to be invested (as reflected by the values of $N$ and $M$) is comparable to that of deterministic test pattern generation. At this point, a list of I-faults with low detection probabilities is produced, to be used as input to the third phase in the test generation process, where more traditional (gate-level) algorithms (e.g., D-algorithm, PODEM or FAN) can be used to increase the fault coverage.

5. RESULTS AND PERFORMANCE

In this section we evaluate WHISTLE from several points of view. First, we illustrate the accuracy of the fault coverage obtained by COVER, by comparing it with the actual fault coverage obtained by gate-level fault simulation. Second, we examine the correlation between the accuracy measurements, as given by our evaluation functions, and the behavior of various characterization functions for a given block. This will illustrate how critical it is to choose a good characterization function, and how it affects the accuracy of the results. A third aspect of WHISTLE that we look at, is its test generation capability, as implemented by BRTPG.

5.1 Accuracy of COVER

For this task we created several combinatorial designs, using a library of six different functional blocks. These blocks are similar to those used by LSI Logic Corporation in their macrocell designs [11], and include a 2-to-4 decoder, an 8-to-1 multiplexor, an 8-to-3 encoder, a 4-bit carry lookahead adder, a 2-bit binary full-adder, and a 4-bit comparator. The designs were "randomly" generated, i.e., the types of blocks and their interconnection were chosen using a random function. Next, the designs were expanded into their gate-level equivalents, by substituting each functional block with its description at the gate level. By assuming a single-stuck-at fault model, DMAT used these descriptions to obtain the necessary information for each block in the design library. Table 3 below summarizes the characteristics of four sample designs, from among those we examined. Notice the ratio of F-faults to I-faults of 1.10 in each case, and also the difference in the number of components (blocks or gates) for each design (approximately 1.20).
### Table 3. Characteristics for the Sample Designs.

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Number of Functional Blocks</th>
<th>Number of Gates</th>
<th>Number of F-faults</th>
<th>Number of I-faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>C899</td>
<td>100</td>
<td>2138</td>
<td>668</td>
<td>8110</td>
</tr>
<tr>
<td>C1185</td>
<td>150</td>
<td>3039</td>
<td>958</td>
<td>11640</td>
</tr>
<tr>
<td>C1690</td>
<td>200</td>
<td>4117</td>
<td>1388</td>
<td>15681</td>
</tr>
<tr>
<td>C2000</td>
<td>250</td>
<td>4940</td>
<td>1702</td>
<td>18672</td>
</tr>
</tbody>
</table>

For each design, we performed functional fault simulation using a set of test patterns, and let COVER estimate the implementation fault coverage. We then compared these results with those obtained by running HSS [18] on the equivalent gate-level designs, using the same stimuli. The results obtained are summarized in Table 4 below, where the `Diff` column shows the difference between our estimates and the actual (from the gate-level fault simulation) fault coverages.

### Table 4. Estimated Implementation Fault Coverage (%).

<table>
<thead>
<tr>
<th>Tests</th>
<th>C899</th>
<th>C1185</th>
<th>C1690</th>
<th>C2000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Est</td>
<td>Diff</td>
<td>Est</td>
<td>Diff</td>
</tr>
<tr>
<td>64</td>
<td>83.0</td>
<td>-1.3</td>
<td>83.6</td>
<td>+0.8</td>
</tr>
<tr>
<td>128</td>
<td>89.3</td>
<td>-1.4</td>
<td>89.9</td>
<td>+0.5</td>
</tr>
<tr>
<td>192</td>
<td>90.9</td>
<td>-1.5</td>
<td>92.0</td>
<td>+0.7</td>
</tr>
<tr>
<td>256</td>
<td>92.0</td>
<td>-1.5</td>
<td>93.5</td>
<td>+0.4</td>
</tr>
<tr>
<td>320</td>
<td>92.9</td>
<td>-1.7</td>
<td>94.4</td>
<td>+0.3</td>
</tr>
<tr>
<td>384</td>
<td>94.2</td>
<td>-1.9</td>
<td>95.1</td>
<td>+0.3</td>
</tr>
<tr>
<td>448</td>
<td>95.1</td>
<td>-1.9</td>
<td>95.5</td>
<td>+0.2</td>
</tr>
<tr>
<td>512</td>
<td>95.5</td>
<td>-1.8</td>
<td>96.3</td>
<td>+0.2</td>
</tr>
<tr>
<td>576</td>
<td>96.6</td>
<td>+0.1</td>
<td>93.6</td>
<td>+0.2</td>
</tr>
<tr>
<td>640</td>
<td>96.8</td>
<td>+0.1</td>
<td>94.1</td>
<td>+0.2</td>
</tr>
<tr>
<td>704</td>
<td>97.2</td>
<td>+0.1</td>
<td>94.7</td>
<td>+0.3</td>
</tr>
<tr>
<td>768</td>
<td>94.8</td>
<td>+0.3</td>
<td>95.4</td>
<td>+0.3</td>
</tr>
<tr>
<td>832</td>
<td>95.5</td>
<td>+0.3</td>
<td>93.3</td>
<td>-0.9</td>
</tr>
<tr>
<td>960</td>
<td>93.6</td>
<td>-1.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From the results in Table 4 we see that the estimate follows very closely the actual implementation fault coverage (less than 2%). We also applied the same procedure to other small examples of hand-coded designs, with similar results, i.e., the estimate differed from the actual fault coverage by at most 2% in either direction.
5.2 Evaluating a Characterization Function

To illustrate the importance of choosing a good characterization function, and how this "goodness" is shown by our evaluation functions, we (randomly) generated a design with 50 blocks of the same type, an 8-to-1 multiplexor. Each such multiplexor has 11 input bits (8 data and 3 control), labeled $i_0 \ldots i_{10}$, and one output bit (i.e., $FF = \{+1, -1\}$). First, using an optimal characterization function for INJENT (i.e., complete accuracy is maintained during the fault-injection process), we study the behavior of COVER for several characterization functions for DETENT. Table 5 below shows these functions and how accurate they are judged to be by the evaluating function $E_{DETENT}$ for each F-fault. Figure 6 shows the fault coverage, as calculated by COVER, using each one of the characterization functions. Also shown is the actual fault coverage obtained using HSS on the gate-level description of the design. We can see that the closer the value of $E_{DETENT}$ is to zero (for each F-fault), the smaller the gap between the fault coverage calculated by COVER, and that obtained by HSS.

<table>
<thead>
<tr>
<th>Characterization Function</th>
<th>Number of Counters</th>
<th>$E_{DETENT}$ $ffv = -1$</th>
<th>$ffv = +1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $i_0 i_1 i_2$</td>
<td>8</td>
<td>0.037</td>
<td>0.063</td>
</tr>
<tr>
<td>2 $i_0 i_2$</td>
<td>4</td>
<td>0.152</td>
<td>0.122</td>
</tr>
<tr>
<td>3 $i_0$</td>
<td>2</td>
<td>0.324</td>
<td>0.382</td>
</tr>
<tr>
<td>4 Uniform Distribution</td>
<td>1</td>
<td>0.701</td>
<td>0.643</td>
</tr>
<tr>
<td>5 $i_1 i_2 \oplus i_3 i_4 i_5$</td>
<td>16</td>
<td>0.702</td>
<td>0.639</td>
</tr>
</tbody>
</table>

Figure 6. Effect of DETENT Characterization Functions on the Fault Coverage

Notice that the fifth function in Table 5 has a domain of 16 values, and thus requires 16 detection counters during simulation. This would seem to promise more accuracy in the results, but, as correctly indicated by our evaluation function, this is not the case (half the number of counters, 8, yields a much better result).
Now let us examine how important an accurate characterization function is for the case of INJENT. To do so, we take the design described above, and use the functions shown in Table 6 to decide when to perform fault injection. The fault coverage results, as calculated by COVER, are shown in Figure 7. We can clearly see the correlation between the value of $E_{\text{INJENT}}$, our evaluating function, and the accuracy of the results obtained by COVER.

### Table 6. INJENT Characterization Functions for 8-to-1 Multiplexer.

<table>
<thead>
<tr>
<th>Characterization Function</th>
<th>$E_{\text{INJENT}}$ for $\text{ffv} = -1$ (or not $\text{ffv} = +1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  $\alpha_0$ and $i_4$</td>
<td>0.887</td>
</tr>
<tr>
<td>2  $\alpha_0$ or $i_{10}$</td>
<td>0.918</td>
</tr>
<tr>
<td>3  $\alpha_0$ or $i_4$ or $i_{10}$</td>
<td>0.985</td>
</tr>
<tr>
<td>4  $i_{10}$</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Figure 7. Effect of INJENT Characterization Functions on the Fault Coverage

### 5.3 Test Pattern Generation by BRTPG

For this series of experiments, we took each of the designs described in Table 3, and used BRTPG to generate a set of test patterns. As in the previous cases, the implementation fault coverage was measured using HSS. The results obtained are summarized in Table 7 below (bold numbers mark the fault coverage achieved by pure random patterns.)
Table 7. Fault Coverage (%).

<table>
<thead>
<tr>
<th>Tests</th>
<th>C899</th>
<th>CI185</th>
<th>C1690</th>
<th>C2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>93.5</td>
<td>93.1</td>
<td>87.9</td>
<td>89.22</td>
</tr>
<tr>
<td>320</td>
<td>94.6</td>
<td>94.1</td>
<td>89.4</td>
<td>90.19</td>
</tr>
<tr>
<td>384</td>
<td>96.1</td>
<td>94.9</td>
<td>90.2</td>
<td>91.16</td>
</tr>
<tr>
<td>448</td>
<td>97.0</td>
<td>95.3</td>
<td>90.6</td>
<td>91.90</td>
</tr>
<tr>
<td>512</td>
<td>97.6</td>
<td>96.1</td>
<td>92.3</td>
<td>92.66</td>
</tr>
<tr>
<td>576</td>
<td>96.4</td>
<td>93.4</td>
<td>92.95</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td>96.7</td>
<td>93.9</td>
<td>93.24</td>
<td></td>
</tr>
<tr>
<td>704</td>
<td>97.1</td>
<td>94.4</td>
<td>93.48</td>
<td></td>
</tr>
<tr>
<td>768</td>
<td>94.5</td>
<td>93.75</td>
<td></td>
<td></td>
</tr>
<tr>
<td>896</td>
<td>95.2</td>
<td>94.20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>960</td>
<td></td>
<td></td>
<td></td>
<td>94.42</td>
</tr>
</tbody>
</table>

5.4 Simulator Performance

To evaluate the performance of HIFAST, we recoded most of its code in S/370 Assembly Language, and then measured the amount of code produced for each sample circuit. We compared these to the original PL/1 code, and to that required by HSS for the same circuits described at the gate-level. (The code measured is the "kernel" of the simulator in each case, i.e., I/O and control portions are not included.) These measurements are summarized in Table 8 below.

Table 8. Code Size (in bytes) for the Sample Designs.

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Gate-Level Code (using HSS)</th>
<th>Functional-Level Code Assembly</th>
<th>Functional-Level Code PL/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>C899</td>
<td>15,676</td>
<td>10,164</td>
<td>71,220</td>
</tr>
<tr>
<td>C1185</td>
<td>23,090</td>
<td>14,350</td>
<td>108,930</td>
</tr>
<tr>
<td>C1690</td>
<td>30,934</td>
<td>19,852</td>
<td>145,514</td>
</tr>
<tr>
<td>C2000</td>
<td>36,974</td>
<td>24,160</td>
<td>179,980</td>
</tr>
</tbody>
</table>

We can see that our Assembly code is about two thirds the size of that generated by HSS. Also, this code is about seven times smaller than the initial (PL/1) prototype code, a proportion which carries over almost unchanged to the execution phase, because of the flatness of the code produced in both cases. Before showing the execution times for the sample designs, Table 9 summarizes the CPU time (on an IBM 4381) spent in the analysis of each block in the library.
Table 9. Time (in seconds) for Block Analysis.

<table>
<thead>
<tr>
<th>Functional Block</th>
<th>CPU seconds (IBM 4381)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-to-4 decoder</td>
<td>0.098</td>
</tr>
<tr>
<td>8-to-1 multiplexor</td>
<td>0.516</td>
</tr>
<tr>
<td>8-to-3 encoder</td>
<td>0.235</td>
</tr>
<tr>
<td>4-bit carry lookahead adder</td>
<td>0.309</td>
</tr>
<tr>
<td>2-bit binary full-adder</td>
<td>0.147</td>
</tr>
<tr>
<td>4-bit comparator</td>
<td>0.170</td>
</tr>
</tbody>
</table>

Table 10 below compares the running time for the functional fault simulator, and HSS, both running on an IBM 4381. Since HSS processes 32 input patterns in parallel, we also run our simulator for 32 patterns, but sequentially.

Table 10. Running Times for 32 Input Patterns (in seconds).

<table>
<thead>
<tr>
<th>Design Name</th>
<th>Gate-Level Fault Simulation</th>
<th>Functional-Level Fault Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C899</td>
<td>9.95</td>
<td>5.40</td>
</tr>
<tr>
<td>C1185</td>
<td>19.75</td>
<td>12.10</td>
</tr>
<tr>
<td>C1690</td>
<td>32.80</td>
<td>20.39</td>
</tr>
<tr>
<td>C2000</td>
<td>45.00</td>
<td>31.60</td>
</tr>
</tbody>
</table>

Considering that the initial version of HIFAST produces straightforward sequential code, without any parallelism or other optimizations, we view these first results as very encouraging. This, coupled with the other advantages offered by addressing the tasks associated with testing at an early stage of the design cycle, make WHISTLE an interesting avenue for further developments.

ACKNOWLEDGEMENTS

We would like to thank I. Berger, of the IBM Haifa Scientific Center, for facilitating the development of the project described in this paper. We also acknowledge the work done by M. Hellman on the implementation of DMAT and the backtracing routines in BRTPG.

6 APPENDIX - PL/ENT Grammar

Declaration Statement

```
VAR variable list
```

Every variable (except variables I and O) must be declared. If no length is specified, the default is 1. The declaration statement may appear anywhere in the program, even after the use of the variables declared in it.
**Constant**
A number between quotes is interpreted as a binary string, otherwise as a decimal number.

**Unary Operators**
- NOT , -

**Boolean Operators**
- XOR, OR, AND
- BOOL(var1, var2, truth-table)
  BOOL returns a bit string that is the result of a Boolean operation, specified by truth-table on the variables var1, var2.

**Arithmetic Operators**
- +, *, -, /

**Return Statement**
- RETURN expression
  The value of the expression will be returned by the function. Notice that when the program reaches a return statement, the rest of the program is not evaluated.

**If Statement**
- IF condition THEN statement(s) FI
- IF condition THEN statement(s) ELSE statement(s) FI
  Every If statement must be terminated by the keyword FI. This allows for a block of statements to be used in either the then-part or the else-part, without using a block "bracketing" structure (e.g., Begin ... End in Pascal).

7 REFERENCES


