A Tool for Measuring Quality of Test Pattern for LSIs' Functional Design

Takashi Aoki
aokit@aecl.ntt.jp

Tomoji Toriyama
tori@center.nel.co.jp

Keiji Ishikawa
ishikawa@aecl.ntt.jp

Kennosuke Fukami
fukami@aecl.ntt.jp

NTT LSI Laboratories
3-1 Morinosato Wakamiya, Atsugi, Kanagawa. 243-01 JAPAN

Abstract— A prototype tool is developed for measuring the quality of test patterns for simulation to verify LSI functional designs. The tool is able to count activated conditional branches and evaluate the branch pass index of test patterns. The branch pass index indicates the ratio of the number of conditional branches validated by the pattern to the total number of conditional branches in a design. We developed the prototype tool for *PARTHENON*[1]. The tool prints out branch identification names not examined by the test pattern. In using the tool for experimental designs, it helped designers to significantly improve pattern quality if a branch pass index of 100% for LSI verification patterns was not achieved. Only about 30 seconds of the processing time was required for a 1000 sentence module. Bugs can often be found in designs with little effort.

I. Introduction

As a higher system performance is required, there are more cases where the functional design of LSIs is translated from such algorithms that were operated on software into a hardware description language (HDL). Even if an algorithm is described as a kind of software language, and if it can be verified through simulation by a test pattern evaluating its quality, we can not always apply a test pattern to the verification of the LSI design successfully because of structure differences in design between software and hardware description. Some differences might be caused by grammatical differences between the software language and HDL, and other differences might be introduced by the designer. As a result, designs that describe some algorithms in HDL must be verified by measuring the quality of hardware verification test patterns. There are only a few methods available in hardware functional designs to count executions of each line in design description[2]; however, they do not clearly take account of two instances produced from the same module. Thus detecting the absence of verification test patterns takes a long time, as dose the problem of locating bugs in the later design stages.

Therefore, we developed a prototype tool for measuring the quality of test patterns for hardware functional simulation to verify LSI designs. In experimental designs the prototype identified deficiencies in the quality of verification test patterns, and helped designers to detect the absence of test patterns.

II. Method

There are several kinds of verification pattern quality indices in the field of software engineering[3]. One of them, the C1 index, is the quality of a test pattern that verifies a software module. It is defined as the ratio of the number of conditional branches examined by the test pattern to the total number of branches in the module.

To measure a kind of C1 quality of verification patterns on a piece of hardware, we modified the hardware description with a strobe register at every conditional branch. Here, the strobe register makes a record of the event each time a branch is examined. When these strobe register values are collected after the simulation, a kind of C1 index can be measured as the ratio of the number of strobe registers which have recorded examinations to the total number of strobe registers. We define this C1 index as a "branch pass index".

The validity of the branch pass index was evaluated against a previous design with around 4000 gates[4] before we started to develop the tool. We made test patterns which have a 100% branch pass index against the design. We found that these patterns satisfy the verification items which we previously made for the design. Therefore, we decided to develop a prototype tool for measuring the branch pass index with this method.

Instead of using a strobe register, the branch pass index can be measured by remaking a simulator so that it can record branch activities. However, this is difficult. On the other hand, since our method is almost completely free from simulator detailed implementation, measurement tools are easily developed and can be applied to some other hardware functional simulator systems.

Moreover, because this method makes each instance has its own strobe registers, even if the hardware described has two instances copied from the same module, the branches in these two instances can be distinguished

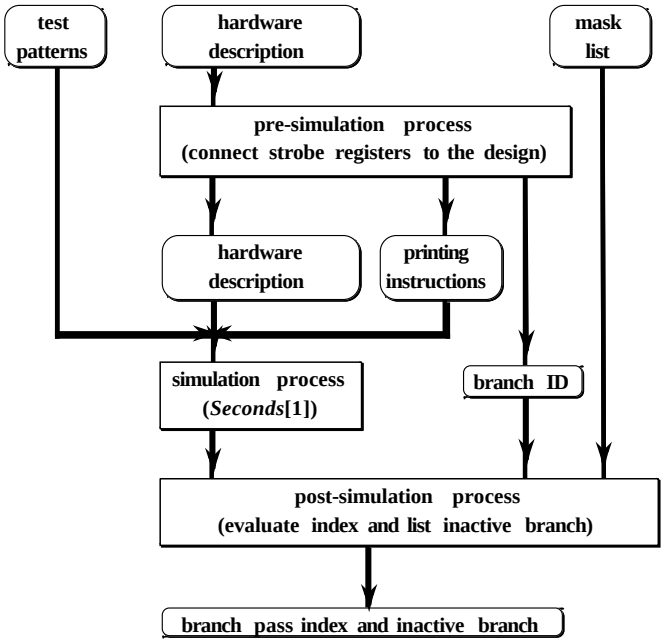

Fig. 1 Structure of prototype tool

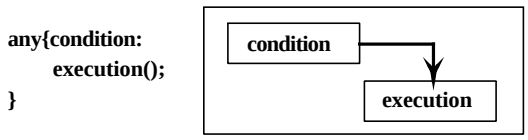

Fig. 2 A hardware description (*SFL*) without a strobe register

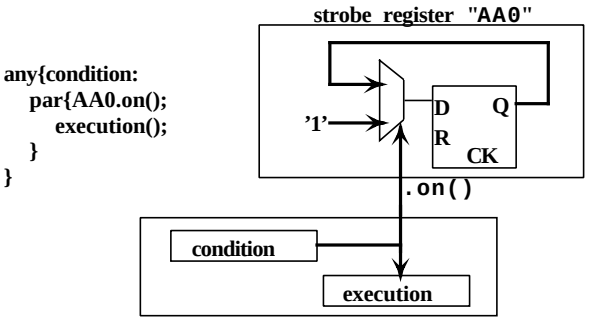

Fig. 3 A hardware description (*SFL*) with a strobe register

unlike the case for measuring a software C1 index. The branch pass index also takes account of conditional state transitions if they are described using conditional branch phrases.

III. A PROTOTYPE TOOL

The tool finds every conditional branch and connects the strobe register to them. It also gives each branch an identification name and reports a branch identification name if the branch is not examined. To make it more useful, the tool is controlled by an operator instruction to suppress reports about branches that were never examined because of their specifications. Here, the operator is the designer who designed and is verifying the hardware. The type of HDL to be processed is SFL[5].

The structure of the prototype tool is shown in fig.1. The designer prepares a hardware functional description in SFL and a verification test pattern. These are the same as for an ordinary simulation. Together with these, the designer can give the tool a branch mask list instructing it to suppress any report about the branches in the list.

The tool consists of two parts: a pre-simulation process and a post-simulation process.

The pre-simulation process generates a hardware description with strobe registers in it. It also generates strobe printing instructions and a branch identification list.

In analyzing the hardware descriptions, whenever the tool finds a conditional branch, it connects a strobe register to the branch and gives the strobe register a local name inside the module, for example a sequence of numbers. Thus each strobe register is connected to a conditional branch.

The strobe printing instructions instruct the simulator to print every value of the strobe register. The values consist of print instructions of the simulator and all strobe register identification names.

Strobe register identification names consist of the hierarchical names of the instances and the strobe register local names in the module. Thus, two register identification names in two instances produced from a module can be distinguished by their names even if they are related to the same branch in the module.

The branch identification list is a list of all branch identification names. A branch identification name is uniquely related to a strobe register identification name that is connected to the branch. Thus, two branch identification names in two instances produced from a module can be distinguished.

Fig.2 is an illustration of a hardware description before a strobe register has been inserted into the conditional branch, and fig.3 is the same part with a strobe register, at the second line "AA0.on()". Fig.4 illustrates the relationships between a test pattern (TP1) and strobe register values. The TP1 activates a dotted line on a flow chart, and it changes the values of S2 and S3 from zero to one. Thus, branches related to S1 and S4 are inactive at TP1.

The simulation is done using *Seconds*[1], a simulator for SFL. The operator gives the simulator the description of the hardware with the strobe registers, and a test pattern with which the hardware description can be verified. At the end of the simulation, the operator gives the strobe

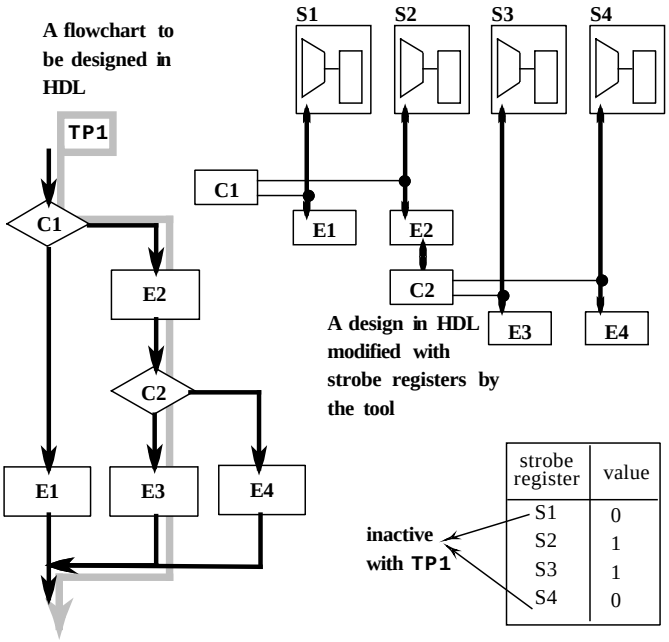

Fig. 4 Relationships between a test pattern (TP1) and strobe register values

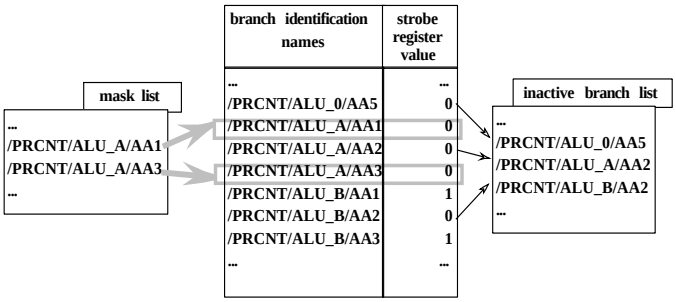

Fig. 5 Suppression of the inactive branch list

printing instructions to the simulator, and makes the simulator print the value of each strobe register.

In the post-simulation process, the tool first extracts the values of the strobe registers. Next, it calculates the branch pass index of the test pattern. Since we use registers with an initial value of zero, the ratio of the number of strobe registers with a non zero value to the total number of strobe registers indicates the branch pass index. Next, it matches each branch identification name to each strobe register value. Branch identification names that match to a value of zero are listed in an inactive branch list. In addition, the tool avoids listing a branch identification name in the inactive branch list if any part of the branch indication name belongs to the branch mask list. Fig.5 illustrates the suppression of the inactive branch list. Consequently, the tool measures the branch pass index, and lists and prints out inactive branches.

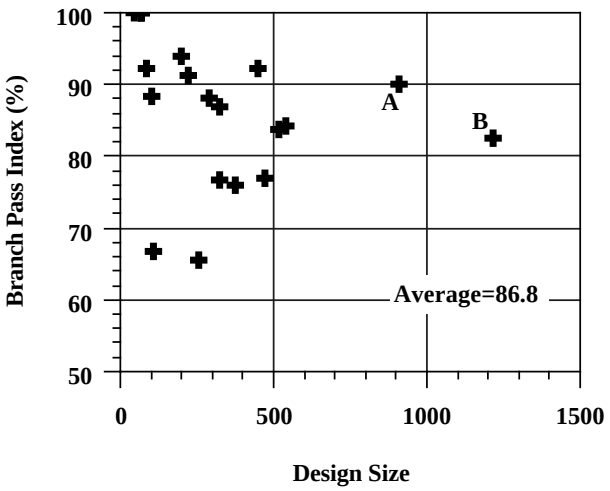

Fig. 6 Measured branch pass indices

IV. Experimental

We applied the tool to the design of LSIs. In our experiment, three designers designed and verified parts of an LSI for use in ATM communication processing using test patterns that they had made.

First of all, each designer designed and verified LSI parts using conventional methods. Thus after the number of bugs accumulated had been stabilized, they made verification test patterns and used them to verify the designs they had made. Next, they measured the branch pass index of their verification test patterns. Finally, each designer used the tool to improve the verification test pattern they had made until the pattern became a 100% branch pass index.

V. Results

Each mark in fig.6 indicates the design size of a module and the branch pass index of its verification test pattern. The design size indicates the number of sentences in the module. The branch pass index was measured using the tool, and was found to be 86.8% on the average.

In the design represented by the letter "B" in fig.6, the pre-simulation process found 236 branches. Only about 30 seconds of pre-simulation and post-simulation processing time was required for this module which was verified through 40 minutes simulation process, using SparcStation 1+. Pre-simulation and post-simulation processing time was about 5 minutes for another 10000 sentences module which was verified through 150 minutes simulation process.

Next, designers improved verification test patterns represented by the letter "A" and "B" in fig.6, using the inactive list produced by the tool. The design "A" has

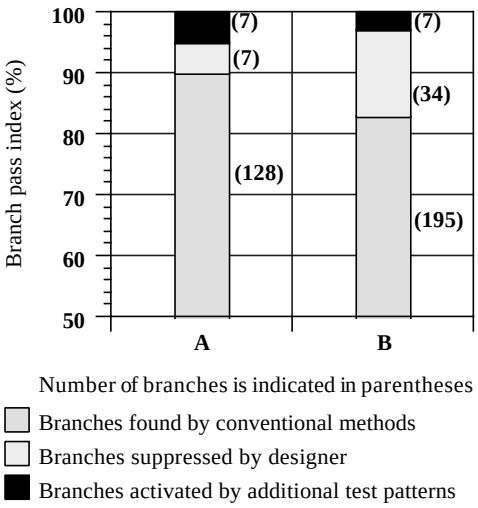

Fig. 7 Improvement possibility using the tool

4.0kG when it is synthesized and 142 branches in its description. This module works to store ATM cell payloads. The module specification is written as a flow-chart with conditional decisions. About 50% of branches used for these decisions, and the other 50% used for arithmetical functions.

The design "B" has 2.9kG when it is synthesized and 236 branches in its description. The module specification is written as block diagrams. About 50% of these branches work as selectors, 25% as decoders, and the other 25% as controllers. This module works to make the ATM cell and send it out.

In the case of sample "A", at first its branch pass index was 90%. The designer effectively made additional test patterns and achieved a 100% branch pass index using only 3% of the overall design and verification time. In this trial 14 inactive branches were found, with 7 ("Branches suppressed by designer" in fig.7) of them inactive because of specification, and 7 ("Branches activated by additional test patterns" in fig.7) of them inactive due to lack of a verification test pattern. Another trial ("B" in fig.6) produced almost similar results. Fig.7 shows the results observed for both A and B.

VI. Discussion

Pre-simulation and post-simulation processing was much shorter than simulation processing time. Simulation time was increased because the simulator processed strobe registers, but we are able to ignore in our experiments.

Of the modules shown in fig.6, only those with less than 100 sentences had 100% branch pass indices in our experiments. In each case of sample "A" and "B", there were 7 inactive branches in the absence of verification at first, when previous verification test patterns were applied to each design.

In the design of "A", there are many combinations of parameters to instruct ways to store ATM cell payloads. These 7 inactive branches are activated on condition that a test pattern is given after some parameter registers are set by another test pattern. Such test pattern relations were absent in sample "A".

In the design "B", there are combinations of signals and states of ATM cell assembly. These all 7 inactive branches are in the absence of test patterns which activate reset signals at several states.

As the design of algorithms becomes more complex, designers become more likely to forget to verify some branches that they have designed, and this may conceal serious problems until later in the design process. Assuming that the size of a design often reflects its complexity, we conclude that in large design cases there is a statistical quality limit to a verification test pattern made solely through the designer's own efforts.

However, improving the verification pattern did not take much time when the designers used this tool. These findings naturally lead us to the conclusion that the tool helps designers to remember branches they had designed previously but had forgotten to validate.

VII. Conclusion

We developed a prototype tool to count executed conditional branches and evaluate verification test pattern quality. In applying the tool to experimental designs, the branch pass index measured was on average 86.8%. The tool helped designers to search inactive branches and to significantly improve pattern quality, even though they could not achieve branch pass quality of 100% for LSI verification patterns.

VIII. Concluding remarks

The quality of test patterns can be measured with the tool when the design is verified by simulation. Consequently, bugs can often be found in designs with little effort. The method does not require remaking simulators and can be applied to some other hardware functional simulator systems. The next stage is to adapt the tool to verilog-HDL and VHDL.

IX. Acknowledgments

The authors would like to thank Dr. Atsushi Takahara for allowing us to use and consult his SFL parser. The authors also wish to thank Dr. Osamu Karatsu, Mr. Yasuyoshi Sakai and Mr. Kazumitsu Takeda for fruitful discussions and advice.

References

[1] Parthenon User's Manual, NTT Data, (1990).

[2] N.2 User's Manual, TD Tech.Inc. (1991).

[3] J. C. Huang, "Program Instrumentation and Software Testing," Computer, 11(4), 25-31, (1978).

[4] T. Toriyama, "A Verification Criteria for LSI Functional Description," The 47th National Convention IPS Japan, C-425 Vol.5 pp.135, (1993 Autumn).

[5] Y. Nakamura, K. Oguri, H. Nakanishi and R. Nomura, "An RTL Behavioral Description Based Logic Design CAD System with Synthesis Capability" Proc. 7th International Conference on Computer Hardware Description Languages and their Applications (IFIP CHDL 85), pp.64-78 (Aug. 1985).