EMULATION TOOL FOR TESTING FAULT TOLERANCE

András P. SOMOGYI

Department of Measurement and Instrument Engineering
Technical University of Budapest
H–1521 Budapest, Hungary
email: somogyi@mmt.bme.hu
Phone and Fax: +36 1 166-4938

Received: September 20, 1993

Abstract

The aim of this paper is to show a possible adaptation of the well known microprocessor emulation method for testing fault tolerance, and to examine the advantages of the adaptation. First a survey on test methods and their application will be given with respect to the possibilities for testing fault tolerant architectures. It will be followed by a short overview of different microprocessor fault models, and the fault injection routines based on the fault models. The injection routines require some hardware extension of the conventional microprocessor in-circuit emulator. The necessary extensions are shown on a MC68000 based in-circuit emulator. Finally, some improvement possibilities are discussed.

Keywords: fault injection, in-circuit emulator, automatic test, fault model of microprocessors.

1. Introduction

Extensive investigations have already been carried out to develop comprehensive and easily adaptable test methods. Also many publications have dealt with various methods of designing fault tolerant computer architectures. Considerably less attention has been directed towards testing fault tolerant architectures. The primary difference between testing fault tolerant and conventional computer architectures is that the correct functioning of an architecture without fault tolerance can be tested through its available I/O ports, while testing a fault tolerant architecture necessitates the access and handling of internal signs.

2. Simulation or Emulation

The various test methods can be arranged into four groups based on their basic strategies: whether the test method uses the existing device under test (DUT) and/or its environment, or whether these components are substituted using simulation.

Normally, simulation assures much easier supervision of the internal points of the system (declared by the model) with respect to both accessing and modifying an internal line. Modifying internal signs of a system requires a more detailed model. The more detailed model used, the more complex supervision can be achieved over the DUT and the more computational capacity is required for executing the test. As a general rule, the function of the DUT containing microprocessor (μP) cannot be efficiently simulated at the μP internal logic level. Therefore simulating its function offers restricted possibility for testing. Since the environment of a DUT is not the target of the test, the environment can be simulated based on its top model.

Testing a system is always carried out either to locate a certain failure, or to check the correct operation of the system. In case of locating a failure and in case of verification, the DUT cannot be simulated. Hence it is unavoidable to develop methods of testing physically existing systems.

The classical ways of a physical test are the 'pin-pad' and the In-Circuit Emulation (ICE) methods. Pin-pad has many disadvantages compared to the ICE, as contact faults, driver problems, different pads for different DUT's, etc. At the same time, ICE can ensure more detailed observability, but only in connection with the emulated part of the DUT. For that purpose, the μP of the DUT is emulated, promising the observability of the most important system parameters.

For the same reason it seems to be obvious to use the μP emulation for injecting fault, first of all into the activity of the μP. Emulation of μP functions makes it possible to emulate faulty processors. To achieve this, the test model of the μP should be discussed.

3. Fault Model of Microprocessors

A fault model description of a μP cannot be built upon the internal physical structure of the μP, since that kind of information is not published. The user-created fault model of a μP can only be established on the basis of the functional description of the μP.

It is, however, obvious that generally used μP's contain nearly the same function units (THATTE et al, 1980). All fault models treated in the literature use these elements as starting-point. These are:

- registers,
- register decoder,
- instruction decoder and controller,
- datapaths and data manipulation units.

Data contained in the registers of a CPU completely specify the state of the processor. The instruction decoder together with the controller are responsible for controlling the transitions between states.

The main emphasis is therefore on the fault model of registers and the instruction decoder. If a register fails to operate correctly, it prevents the processor from storing its state correctly which may cause (in case of special function registers) unrecoverable errors. If the instruction decoder and/or the control unit of the CPU is faulty, it will prevent the processor from executing certain state-transitions. All faults of the datapaths can be mapped onto these units, therefore it is not necessary to deal with the fault model of the internal datapath. The internal datapath structure of the processor is usually unknown, thus it is impossible to describe the effect of an internal bus fault.

Fault models of μP registers and the register decoder are similar to those of memory arrays (NAIR et al., 1978; MARINESCU, 1982): each bitcell of a register can

- have stuck-at-x,
- have stuck-open,
- have transient,
- have multiple access,
- have data retention fault,
- be state coupled to another cell.

Typical register decoder faults are:

- no register selected,
- one or more registers selected instead of the required,
- one or more registers selected in addition to the required.

Generally defined: In case of a register decoder fault instead of a register R_i a set of registers are selected, where the set can be empty, or may, or may not contain the original register R_i. Based on their functional effect the bitcell faults, like multiple access, and state coupled cells cannot be distinguished from decoder faults. Some of these faults can also be data dependent. Our analysis is done only for data-independent faults.

Executing an instruction with a faulty instruction decoder can cause

- no instruction to be executed,
- or another instruction to be executed instead of the required,
- or other instruction(s) to be executed additionally to the required.

This model (THATTE et al., 1980) uses the easily accessible instruction level of the CPU, thus it is easy to use. However, it is not suitable for modelling partially executed instructions. Faults like partially executed instructions occur in all kinds of today's μP's, since executing an instruction in these μP's always means sequential execution of several so-called microinstructions, where executing a microinstruction means parallel execution of more microorders. Fault models dealing with that problem should be based on the possible microorders and microinstructions of the μP (ABRAHAM et al., 1984; FRENZEL et al, 1984). Since no documentation of the CPU on this level is published, determining these collections cannot be done deterministically.

4. Fault Injection of Microprocessors

The faulty operation of the emulated CPU should be simulated according to the discussed fault models. Because of the complexity of the μP's, the μP emulating device usually uses the same type of μP as the original, or a type compatible with it. Through emulating the μP with the same type of processor, the only possible effect level is the instruction level of the μP. All fault models described above should be implemented by using instructions of the processor.

4.1. The Global Algorithm

The global algorithm of simulating a faulty processor is the following:

- While the processor is executing the original instructions of the DUT, the simulator should determine for each instruction, whether the simulated fault does or does not have any effect on the correct instruction. A fault does affect an instruction if the result state after executing the instruction in case of a faulty processor differs from the result state of a fault-free processor (if both processors were in the same state before executing the instruction).
- If the fault does not have any effect on the current instruction, no intervention is needed.
- If the fault does have some effect on the current instruction, the execution should be interrupted in order to ensure saving and preparing the processor state for the faulty execution. This part of the simulation is called injection pre-routine. After the pre-routine the original instruction can be executed. It is possible, however, to omit the original instruction. It will be defined by the implementation of the fault model.
- After executing the pre-routine and the original instruction, another routine should be activated to shape the result state of the processor in accordance with the executed instruction and the simulated fault. This routine is called the post-routine.

Implementing the necessary fault models means defining the pre- and post-routines for each fault and instruction. n instructions and k fault models result in $k \times n$ pre- and post-routines. It will be shown, however, that many of these routines are the same.

4.2. Implementation

The fault model for the μP registers and the register decoder will result in approximately only as many pre- and post-routines as many different types of faults are defined:

- Bitcell faults of register R_i as stuck-at, stuck-open or transient faults: all instructions writing to R_i should be extended by the specific pre- and post-routines. The task of these routines is to compensate the write instruction for the faulty cell. The whole instruction cannot be omitted since it may have effect on other cells. The pre-routine will save the content of R_i before the instruction, and after the execution of the write instruction the post-routine will restore the content of the specific cell.
- Bitcell faults of register R_i as data retention faults: this type of fault can be simulated through executing a specific routine once only, not associated with any instruction, but associated e.g. with time. The task of this routine is to reset the faulty cell.
- Register decoder fault: All types of these faults can be simulated by injecting routines of similar construction. Two sets of injection routines should be implemented. One for all instructions trying to write R_i (i.e. R_i is the destination), and one for all instructions trying to read R_i (R_i is the source). In case of write-type instructions, the pre-routine must save the content of R_i, and the post-routine should restore it and overwrite all the registers affected by the decoder fault. In case of read-type instructions, the pre-routine should save the content of R_i, then overwrite it with a specific logical combination of the contents of the registers affected by the register decoder fault. The post-routine will in that case do not more than restore the original content of R_i.

Special problems arise, however, when simulating a faulty program-counter register. That fault affects each instruction execution, since the PC register is modified after each instruction. It will cause all instructions to be indicated as affected ones. Injecting routines before and after each instruction will slow down the system to an unacceptable extent. This problem can be solved by some hardware extension. A fault affecting all bitcells of the PC-register of the CPU (e.g. the register cannot be selected), is a catastrophic fault from the point of view of the system. If the fault affects only one or some of the bit cells of the PC, then an injection routine is only needed if the faulty cells are changed, hence the μP reads instruction from address matching a given bitmap. If the breakpoint unit of the ICE can catch these cycles, it is unnecessary to interrupt the program execution after each instruction.

The implementation of instruction decoder and control faults is not so obvious. For the basic fault model (when instead of instruction I_i a set of instructions are executed where the set can be empty, or may or may not contain the original instruction I_i) a general implementation can be given: the execution of the original instruction should be omitted, and all the instructions in the set should be executed. (Because of omitting the original instruction, the pre- and post-routines are merged.) For implementing faults causing partially executed instructions, the set of microorders and microinstructions and their effect(s) on the processor state should be determined and described in a library. The information necessary for that can be gathered from the instruction manual of the μP, although it is not described directly (ABRAHAM, 1984). The tasks of the pre- and post-routines for specific fault models can be determined on the basis of this library.

5. Hardware Requirements

The hardware requirement of the method of simulating processor faults as described above is that the emulation of the processor should be interruptable at any arbitrary specified set of instructions.

5.1. The Function of ICE

Let us consider a μP ICE for emulating the processor MC68000 with the same type of processor (SOMOGYI, 1988). μP- ICE's using the same type of processor as that of the emulated one, have got two operation modes: host mode and target mode. The emulation is performed in target mode: the processor executes the instructions fetched from the DUT. If a breakpoint occurs the emulation is aborted, and the ICE is switched to the host mode. In host mode the CPU executes instructions from the ICE memory. This mode is necessary to ensure observability of the processor's internal state. The breakpoint unit of a conventional ICE does not support the possibility of interrupting the emulation at an arbitrary specified set of instructions for two reasons: it would need much more memory, and the identification of the next in turn instruction code based solely on the μP bus sign would be impossible.

Since the MC68000 cannot have more than 65536 instructions, a memory of 64 kbit is needed to store the information whether an instruction should or should not cause breakpoint.

5.2. Identification of the Next Instruction

The identification of the next instruction is prevented by the pipeline feature of the μP, and by the fact that no sign of the μP bus indicates the first word of a fetch cycle.

To eliminate these difficulties the following procedure is applied: Whenever the data bus of the μP contains data equal to the code of an instruction where injection should be done, a non maskable interrupt is performed. The interruption causes the processor to save its state, including the address of the next instruction to be executed. If that address is equal to the address of the data causing the interrupt, then that data is a code of an instruction where injection should be done. In this case, the pre-routine is activated. If the address of the next instruction is not equal to that of the data, then the data is either not an instruction code, or is not the following instruction (pipeline!).

5.3. Other Problems

In case of newly developed processors, such as the 68030, another problem arises. The processor type 68030 contains instruction cache, making the instruction fetch order unobservable. This problem can be solved by separating the fault tolerance test against cache faults and faults in the conventional part of the μP.

6. The User Platform

The user platform of the fault injector supports the implementation of the fault models discussed above. It means that the user need not specify the details of injecting a given fault, but the required fault should be selected from the injection library. Furthermore, it is possible for the user to describe special faults and add them to the library. The description follows the injection method described above: for defining a new fault injection, first the instructions should be listed which are affected by that fault. For simple usage, all indicated instructions use the same pre- and post-routines. For advanced applications, special routines can also be defined for different instructions within the same fault. With this construction, nearly all types of microprocessor faults can be simulated.

After choosing the appropriate fault(s) from the library, the conditions of the injection should be set. The available condition options are derived from the occurrence mode of the real fault. Such options are: transient or permanent faults. In case of transient error the triggering condition: random, time, cycle-time, data, etc.

The evaluation of the experiment is carried out on the basis of the trace diary and with the aid of the fault detection feedback. Since most single processor devices implement fault detection only, where the detection is done by adding some special hardware (e.g. watch-dog), it is necessary to ensure an external feedback between the DUT and the ICE. The collected data are stored in the experiment diary for later (off-line) analysis.

7. Conclusions and Improvement Possibilities

The presented method is developed for testing the fault tolerance of the μP controlled devices against μP faults. For most of the typical faults of μPs, algorithms can be given for injecting these faults. For the remaining cases, a general Mainframe is defined for the user to develop his own fault injection algorithm.

The set of faults which can be injected by the emulator can be extended over other units of DUT connected with the P bus, e.g. memories. It should be noted, however, that this extension is only effective if no fault-masking is performed between the processor and the unit. The effect of all faults of the unit should be mapped onto the μP bus. Special injection routines should be activated around all the instructions which address the unit under discussion.

The developed fault tolerance test tool is suitable not only for final verification of fault tolerance, but can also be effective in the development phase, since debugging is also aided through the trace diary.

References

NAIR, R. – THATTE, S.M. – ABRAHAM, J.A. (1978): Efficient Algorithms for Testing Semiconductor Random-access Memories. *IEEE Transactions on Computer*, June 1978, pp. 572-576.

MARINESCU, M. (1982): Simple and Efficient Algorithms for Functioanl RAM Testing. *IEEE Test Conference* 1982, pp. 236-239.

THATTE, S.M. – ABRAHAM, J.A. (1980): Test Generation for Microprocessor. *IEEE Transactions on Computer*, June 1980, pp. 429-441.

ABRAHAM, J.A. – BRAHME, D. (1984): Functional Testing of Microprocessor. *IEEE Transactions on Computer*, June 1984, pp. 475-485.

FRENZEL, J.F. – MARINOS, P.N. (1984): Functional Testing of Microprocessor in a User Environment. *IEEE Fault Tolerance Conference* 2984, pp. 219-224.

SOMOGYI, A. P. (1988): MC68000 In-Circuit Emulator. Stud. P. Contest of TUB, 1988.