IMPLEMENTATION OF A 16 BIT MEMORY UNIT USING SABOTEURS AND MUTANTS

SUSRUTHA BABU SUKHAVASI[1*], SUPARSHYA BABU SUKHAVASI[1]
DR.HABIBULLA KHAN[2] , CHIRANJEEVI PILLA[3]
1*.ASSISTANT PROFESSOR, Department of ECE, K L University, Guntur, AP, India
1. ASSISTANT PROFESSOR, Department of ECE, K L University, Guntur, AP, India
2. PROFESSOR & HEAD, Department of ECE, K L University, Guntur, AP, India
3. M.TECH-VLSI STUDENT Department of ECE, K L University, Guntur, AP, India.

ABSTRACT:

Fault tolerant circuits are currently required in several major application sectors, and a new generation of CAD tools is required to automate the insertion and validation of fault tolerant mechanisms. This paper outlines the characteristics of a new fault injection platform and its evaluation in a real industrial environment and also presents a technique to improve verification at the VHDL level of digital circuits by means of a specially designed fault injection block.

Fault injection techniques based on the use of hardware description languages offer important advantages with regard to other techniques. First, as this type of techniques can be applied during the design phase of the system, they permit reducing the time-to-market. Second, they present high controllability and reach ability. Among the different techniques, those based on the use of saboteurs and mutants are especially attractive due to their high fault modelling capability. However, implementing automatically these techniques in a fault injection tool is difficult. Especially complex are the insertion of saboteurs and the generation of mutants. In this paper, we present new proposals to implement saboteurs for models in VHDL which are easy-to-automate, and whose philosophy can be generalized to other hardware description languages.

1. INTRODUCTION:

The importance of fault tolerance (FT) of computing systems is increasing instantly nowadays. This is a consequence of the technology trends which try to follow Moore’s law in increasing chip density by decreasing feature size. Smaller feature size, greater chip density, and minimal power consumption lead to increasing device vulnerability to external disturbances such as radiation, internal problems such as crosstalk, and other reliability problems, which result in an increasing number of faults, especially transients, in computing systems.

Fault injection is a validation technique of fault tolerant systems (FTSs) which is being increasingly consolidated and applied in a wide range of fields, and several automatic tools have been designed. Fault injection is defined in the following way. “Fault injection is the validation technique of the Dependability of Fault Tolerant Systems, which consists in the accomplishment of controlled experiments where the observation of the system’s behaviour in presence of faults is induced explicitly by the written introduction (injection) of faults in the system”. This analysis can be either the study of the incidence of faults on the system (called Error syndrome analysis) or checking the design specifications (called Validation).

The objective of the error syndrome analysis is to detect those parts of the system which are most sensitive to faults, and eventually, to choose the most suitable fault-tolerance mechanisms (FTMs). The aim of the validation is to verify that the system and/or its built-in FTMs accomplish the design specifications in presence of faults. If the dependability is analyzed at early phases of the design cycle, both time and money can be saved in the development process. A common experimental method to validate the dependability of a fault tolerant system (FTS) is fault injection, which is defined in as the deliberate introduction of faults into a system (the target system).

There are works related to fault injection with saboteurs and mutants in other areas like test or field programmable gate array (FPGA)-based fault emulation, although the objective of the study in each area is quite different. In dependability analysis, the objective can be either to verify the sensitivity to physical faults or validate the effectiveness of the FTMs of a simulation model of the system under analysis, by modifying the operation of the model at simulation time. In test, the aim of fault injection is to accelerate the test process by obtaining reduced test pattern lists injecting faults at higher abstraction levels, like register- transfer (RT) or system.

The main motivations and goals for this research concern is the development of an integrated design environment. The expected benefits of such an environment with respect to
(i) The analysis of the fault activation and error propagation processes

(ii) The guidance of the fault injection process according to the validation objectives, are clearly identified.

This provides substantial motivations for the choice of the simulation language.

2. RELATED WORK:

The first step in a modern digital system design is to specify it in a high level language such as VHDL. Before the translation of the specification into an actual implementation, the design needs to be evaluated based on several criteria, e.g. area, testability, power consumption etc. The capability to verify a testable system (in the presence of faults) at the VHDL level before it is implemented allows design modifications to achieve the desired goal. This makes the case for a fault injection system that provides such capability. In general faults are separated into two categories: permanent and transient. Permanent faults that exist in logic circuits are normally identified during offline testing by the manufacturer of the IC, so the transient fault is of major concern after a chip is in the hands of the consumer. The ability to simulate the occurrence of a transient fault in the VHDL description of a system is extremely important to verify the performance of an on-line testable system In addition the ability to insert permanent faults on single bits or a data word must also be taken into consideration. These features enable the performance of a system under faulty conditions to be effectively verified before the system is implemented.

Generally, Fault injection is commonly used for the validation of the fault tolerance as it can be viewed as a test of the FTAMs with respect to specific inputs: the faults. However, these activities make use of specific methods and tools that are somewhat disconnected from those applied in the design; this is particularly true when considering fault injection on a prototype of the target system.

A simulation environment provides enhanced controllability and observability on the target system. This will improve the flexibility of application of fault injection (e.g., with respect to the mastering of the synchronization of the fault injection with the operational activity on the model of the target system).

2.1. Guidance of the fault injection process:

Two main objectives can be identified for the fault injection experiments to be carried out:

• Fault removal, i.e., the correction of potential fault tolerance deficiencies in the FTAMs

• Fault forecasting, i.e., the evaluation of the coverage distribution (e.g., coverage factor and latency) provided by the tested FTAMs.

In both cases, the efficiency and relevance of the fault injection experiments should be maximized. This encompasses both faults injected in and activation provided to the target system. Regarding the fault removal objective, the test should be directed to achieve a high coverage of the possible configurations of the FTAMs to be validated on the target system. In this case, the selection of the faults/errors to apply and errors to propagate is primarily based on the analysis of the model describing the FTAMs and of the information flow in the simulation of the FTAMs. In practice, it may be useful to rely on a predefined and limited set of error classes (those corresponding to the design assumptions of the FTAMs). References and have addressed this issue in the context of fault-tolerant protocols.

3. FAULT INJECTION TECHNIQUES:

Engineers use fault injection to test fault-tolerant systems or components. Fault injection tests fault detection, fault isolation, and reconfiguration and recovery capabilities.

3.1. Fault injection environment:

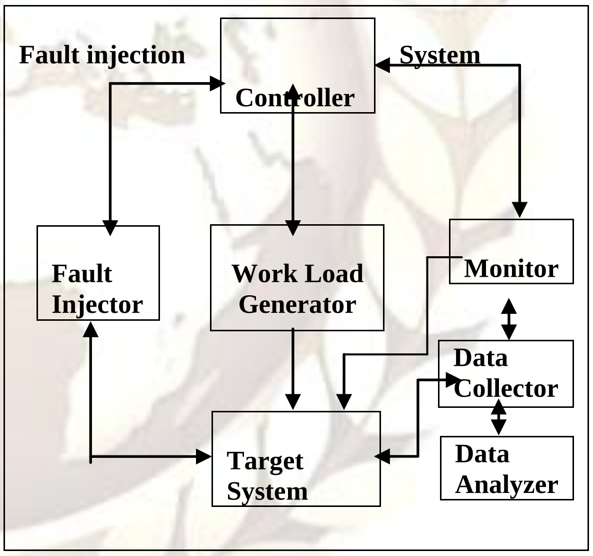


Figure 3.1 Basic components of a fault injection environment

Fig.3.1 shows a fault injection environment, which typically consists of the target system plus a fault injector, fault library, workload generator, workload library, controller, monitor, data collector, and data analyzer. The fault injector injects faults into the target system as it executes commands from the workload generator (applications, benchmarks, or synthetic workloads). The monitor tracks the execution of the commands and initiates data collection whenever necessary.

The data collector performs online data collection, and the data analyzer, which can be offline, performs data processing and analysis. The controller controls the experiment. Physically, the controller is a program that can run on the target system or on a separate computer. The fault injector can be custom-built hardware or software. The fault injector itself can support different fault types, fault locations, fault times, and appropriate hardware semantics or software structure—the values of which are drawn from a fault library. The fault library in Figure 1 is a separate component, which allows for greater flexibility and portability. The workload generator, monitor, and other components can be implemented the same way.

VFIT, a VHDL - based fault injection tool that applies several of the previously described techniques. In fact, only the *other techniques* group has not been implemented due to their excessive complexity. VFIT (VHDL - based fault injection tool) that runs on PC computers (or compatible) under Windows and is model-independent. Although it admits models at any abstraction level, it has been mainly used on models at gate and RT levels.With VFIT it is possible to inject faults automatically applying the simulator commands technique. It is also feasible to inject faults using saboteurs and mutants, but in this case, the injection process needs the intervention of the user because the insertion of the saboteurs and the generation of mutants are not automatic.

3.2. Classifying Fault Injection Techniques:

The fault injection is a technique of Fault Tolerant Systems (FTSs) validation which is being increasingly consolidated and applied in a wide range of fields, and several automatic tools have been designed. The fault injection technique is defined in the following way:

Fault injection is the validation technique of the Dependability of Fault Tolerant Systems which consists in the accomplishment of controlled experiments where the observation of the system's behavior in presence of faults is induced explicitly by the writing introduction (injection) of faults in the system. The fault injection techniques in the hardware of a system can be classified in three main categories:

3.2.1. Physical fault injection (HWIFI):

It is accomplished at physical level, disturbing the hardware with parameters of the environment (heavy ions radiation, electromagnetic interferences etc.) or modifying the value of the pins of the integrated circuits. Hardware-implemented fault injection uses additional hardware to introduce faults into the target system's hardware. Depending on the faults and their locations, hardware-implemented fault injection methods fall into two categories:

• ***Hardware fault injection with contact***:

The injector has direct physical contact with the target system, producing voltage or current changes externally to the target chip. Examples are methods that use pin-level probes and sockets.

• ***Hardware fault injection without contact :***

The injector has no direct physical contact with the target system. Instead, an external source produces some physical phenomenon, such as heavyion radiation and electromagnetic interference, causing spurious currents inside the target chip. Hardware fault injections occur in actual examples of the circuit after fabrication. The circuit is subjected to some sort of interference to produce the fault, and the resulting behavior is examined. So far, this has been done with transient faults, as the difficulty and expense of introducing stuck-at and bridging faults in the circuit has not been overcome.

The circuit is attached to a testing apparatus which operates it and examines the behavior after the fault is injected. This consumes time to prepare the circuit and test it, but such tests generally proceed faster than simulation does. It is, rather obviously, used to test circuit just before or in production. These simulations are non-intrusive, since they do not alter the behavior of the circuit other than to introduce the fault. Should special circuitry be included to cause or simulate faults in the finished circuit, these would most likely affect the timing or other characteristics of the circuit, and therefore be intrusive.

3.2.2. Software Implemented Fault Injection (SWIFI):

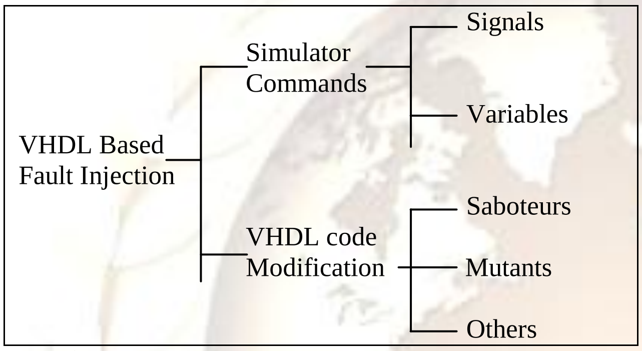
The objective of this technique, also called Fault Emulation, consists of reproducing at software level the errors that would have been produced upon occurring faults in the hardware. It is based on different practical types of injection, such as the modification of the memory data, or the mutation of either the application software or the lowest service layers (at operative system level, for example).

Software fault injection is used to inject faults into the operation of software and examine the effects. This is generally used on code that has communicative or cooperative functions so that there is enough interaction to make fault injection useful. All sorts of faults may be injected, from register and memory faults, to dropped or replicated network packets, to erroneous error conditions and flags. These faults may be injected into simulations of complex systems where the interactions are understood though not the details of implementation, or they may be injected into operating systems to examine the effects. Software fault injections are more oriented towards implementation details, and can address program state as well as communication and interactions. Faults are mistimings, missing messages, replays, corrupted memory or registers, faulty disk reads.

The system is then run with the fault to examine its behavior. These simulations tend to take longer because they encapsulate all of the operation and detail of the system, but they will more accurately capture the timing aspects of the system. This testing is performed to verify the system's reaction to introduced faults and catalog the faults successfully dealt with. this is done later in the design cycle to show performance for a final or near-final design.

3.2.3. Simulated fault injection:

In this technique, the system under test is simulated in other computer system.



The faults are induced altering the logical values during the simulation. Some paper describes a tool for injecting faults in VHDL simulation models.

This work is framed in the simulated fault injection, and concretely in the simulation of models based on the VHDL hardware description language. We have chosen this technique due fundamentally to:

1. The growing interest of the simulated injection techniques as a complement of the physical fault injection (these have been traditionally more numerous and developed) and Fault Emulation (SWIFI). The greatest advantage of this method over the previous ones is the Observability and Controllability of all the modelled components. The simulation can be accomplished in different abstraction levels. Another positive aspect of this technique is the possibility of carrying out the validation of the system during the design phase, before having the final product.

2. The good perspectives of modelling systems and faults with VHDL that has been consolidated as a powerful standard to analyse and design computer systems.

3. Observe system behavior in the presence of faults.A fault is a random or malicious defect introduced to the system. A fault may cause an error state of the system. A system enters error state if its normal operation can not be performed anymore (due to a fault). A recognized error does not mean a failure of the system.

The system fails if it no longer meets the requirements for proper functions.Fault tolerant systems are used in safety critical applications. Fault tolerant (FT) system – a system that provides required functionality even in the presence of faults.Safety critical application – the cost of a failure is much higher than the price of the system, e.g. human lives are in danger, a production plant is stopped. Real-time (RT) system – the system responds to events immediately as they occur. Hard RT systems provide guaranteed deadlines. Type of fault is a class that have its own constructor and attributes. A fault is an instance of this class.

Simulation-based fault injection is a useful experimental way to evaluate the dependability of a system during the design phase, thus reducing the time-to-market. Another interesting advantage of this group of techniques with regard to others is that those based on simulation offer both high observability and controllability of all the modeled components.

3.2.4. Parameters of the Injection Campaign:

The following list describes the fault injection experiment conditions of the injection Campaign of this work.

I. Number of faults:
n = 3000 single faults in every injection campaign. This guarantees the statistical validity of the results.

II. Workload:
The workload is a simple program that obtains the arithmetic series of n integer numbers.

III. Fault types:

The injected faults are transient/permanent, stuck-at 0, stuck-at 1, Open-line, or indetermination and they may affect the signals and the variables in the model.

IV. Place where the fault is injected:
The faults are systematically injected on any atomic signal (sets of signals, like buses, are divided into their bits) and variable of the model, in both the external structural architecture and the behavioural architectures of the components. Faults are not injected in the spare CPU, since it is off while the system is working properly.

V. Values of the faults:
The values of faults are produced according a Uniform distribution along the available range of signals and variables.

VI. Time instant when the fault is injected:
The time instant of injection is distributed according to different types of probability distribution functions

(Uniform, Exponential, Weibull, and Gaussian), typical of the transient faults, in the range:

$[0, t_{workload}]$;

Where $t_{workload}$ = workload simulation duration without faults.

VII. Simulation duration:

The simulation duration includes the execution time of the workload and the recovery time with the spare CPU ($t_{simul} = t_{workload} + t_{spare}$). In our case, we have chosen a value of 500mS.

4. VHDL-BASED FAULT INJECTION TECHNIQUES:

There exist a group of fault injection techniques based on the use of hardware description languages (or HDLs) as modeling languages. The most popular high-level HDLs are VHDL, Verilog, and SystemC. In our case, we work with VHDL . These techniques are widely applied, due to the advantages of employing an HDL. The present work is framed in this group of techniques. Classification of VHDL-based fault injection techniques, Nevertheless, both this taxonomy and the description of the injection techniques can be generalized to any other HDL.

4.1.Fault Injection Using Simulator Commands:

This fault injection technique is based on using the commands of the simulator at simulation time, in order to modify the value or timing of the signals and variables of the model. Using simulator commands it is possible to inject transient, permanent, and intermittent faults. Though, there exists one restriction: due to the special nature of variables in VHDL, it is not possible to inject permanent faults in variables. This technique is the easiest one to implement and its temporal cost (to perform the simulation) is by far the lowest. However, the number of fault models that can be injected is smaller than with the other techniques

4.2. Techniques for injecting faults into VHDL models:

Three types of techniques for fault injection can be identified: those that require modification of the VHDL code and those that use the built-in commands of the simulator. These techniques are described and compared in this section.

4.2.1 Modification of the VHDL model:

Two techniques can be distinguished. The first is based on the addition to the VHDL model of dedicated fault injection components, called saboteurs. The second is based on the mutation of existing component descriptions in the VHDL model, which generates modified component descriptions called mutants. A saboteur is a component added to the VHDL model for the sole purpose of fault injection. It is inactive during normal system operation, while altering the value or timing characteristics of one or more signals when active, i.e., when a fault is being injected. Saboteurs are inserted, either interactively at the schematic editor level or manually/automatically directly into the VHDL source code.

A serial and a parallel technique of insertion can be distinguished. Serial insertion, in its simplest form, consists of breaking up the signal path between a driver (output) and its corresponding receiver (input) and placing a saboteur in between. In its more complex form, it is possible to break up the signal paths between a set of drivers and its corresponding set of receivers and insert a saboteur. Using the latter form, the signal value for a receiver can be a function of the values provided by the set of drivers, allowing complex faults to be modelled, e.g., signal crosstalk. For parallel insertion, a saboteur is simply added as an additional driver for a resolved signal.

A mutant is a component description that replaces another component description. It behaves as the component description it replaces, except during fault injection when the mutant's behaviour is switched to imitate the component's behaviour in presence of faults. It is easy to implement this technique in VHDL when the mutant is distinguished from the original component description by a different architecture, as the configuration mechanism can be used to select an architecture (in our case the mutated one) for the component.

The required mutation may be accomplished in several ways by:

• adding saboteur(s) to structural or behavioral component descriptions.

• Mutating structural component descriptions by replacing subcomponents; for example, a NAND-gate may be replaced by a NOR-gate.

• Automatically mutating statements in behavioral component descriptions, e.g., by generating wrong operators or exchanging variable identifiers; this approach is similar to the mutation techniques used by the software testing community.

• Manually mutating behavioral component descriptions to achieve complex and detailed fault models.

In all cases, it simplifies matters if the mutant has an interface (entity declaration) compatible with the component description it replaces. Indeed, the VHDL code for the component description in which the mutant itself is a subcomponent may remain unchanged.

4.3. Fault injection using a behavioural model:

4.3.1. Bit-flip injection using a VHDL description:

Using behavioral descriptions to study the consequences of faults for complex circuits has been widely proposed in the specialized literature. A review of VHDL-based techniques can be found .Here we have applied the so-called "saboteurs" technique to a VHDL behavioral model of 80C51 in order to study the effects of bit flips when executing a program.

The VHDL model uses an array of 8 bit vectors in order to simulate all the 128 internal RAM bytes and Special Function Registers (SFR) included in the 8051 architecture. Series of tests were performed where faults were randomly injected both in location of the affected bits inside this array and in time of the SEU occurrence. Note that injected faults did not target the memory bits of program code to be executed by the micro controller, the fault injection being performed by means of suitable modifications added to the VHDL signals within the emulated 8051.

The setup of the experiment was therefore a simulation setup. In fact, we generated a VHDL schematic with the instantiation of the studied micro controller and other needed blocks. The main blocks are:

- 8051
- SRAM 64k
- ROM 4k

These blocks were simulated using a commercial VHDL simulator. The only modification we made to the VHDL model was to add a saboteur process able to inject bit flips inside the registers within the 8051.The VHDL simulator, concurrently with the normal processes emulating the 8051 micro controller, executes the saboteur process.

Therefore the activation of fault injection, performed by means of the two extra signals IND and BIT, is totally independent and asynchronous to the state of the 8051.

In fig 4.3.1.1 is shown a schematic description of the fault injection strategy in a general case. The VHDL behavioral description of the 8051 can be seen as a set of concurrent processes, each one implementing a function of the micro controller (ALU, PC incrementer, Watchdog, etc.) and the communication between these processes is provided by a set of internal signals visible to all the processes. Some of these signals have physical counterparts like SFR, RAM, PC and others. The saboteur is a special process that runs concurrently to the other processes and is activated, in our case, by two external commands IND and BIT.

When normal operation (without fault injection) is carried out the saboteur is in a stand-by mode, when the fault injection is activated, the saboteur modifies an internal signal inverting its value. In this way the saboteur provides an asynchronous SEU injection on any internal signal of the VHDL description.

It has to be noticed that, in order to have a realistic behavior of the micro controller, the injection must be performed only on those signals representing physical registers. Assuming that the SEUs mainly affect the internal registers rather than combinatorial logic, we isolated the signals representing these registers and made them our SEU injection target. The fault injection technique is composed of the following steps.
First is defined the time width of injection zone relatively to the program that must be tested, secondly is defined the number of logical targets that must be used in order to inject error in all internal register (in this case 152).

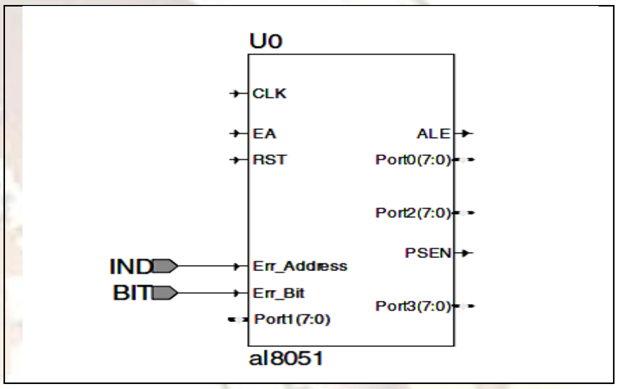


Fig4.3.1.1 Saboteur process

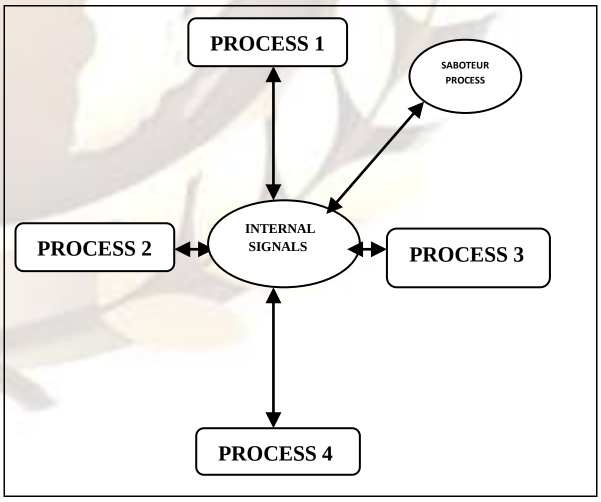


Fig4.3.1.2. Structure of VHDL 8051 model with fault injection capability

In the above figure the external 8051-module instantiation with fault injection capability. The IND and BIT signals appear as two additional ports of the 8051 and are driven by the simulator executing a macro file.IND Signal indicates the address of the internal byte to be affected by the injected fault and BIT indicates Specific bit to flip.
The instance of injection is forced by the macro together with these two addresses.

4.3.2. Setup of a VHDL fault injection Campaign

Setting up a fault injection campaign requires the following steps:

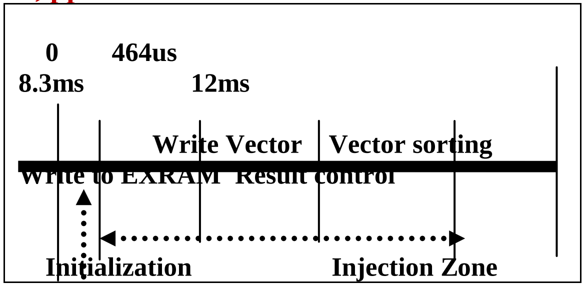
- Analysis of the VHDL model
- Set up of the Saboteur process
- Set up of the Macro generator Program

In principle these steps can be applied to any VHDL description of a digital system including memory elements. For instance, can be corrupted by this method the content of bits of a Finite State Machine status register, as well as the internal registers of a micro controller description. The first step requires the analysis of the VHDL description to find the targets suitable for bit-flip injection. The second step is done adding the saboteur VHDL process described above, capable of modification of the value of the selected target.

The third step is the generation of the macro file that drives the simulator engine giving both the stimuli to the VHDL description of the device under test (like clock reset etc) and the randomly generated activation stimuli for the saboteur process. The first step is the more important change to be afforded when a new device VHDL description must be tested. Once the targets of the VHDL code are identified, the modifications related to the second step consist in the connection of the saboteur to the selected target. The stimuli for the macro generator to the saboteur are almost the same while the stimuli for the DUT VHDL description are strictly related to its functions.

Therefore, once the setup phase is performed the fault injection campaign can be carried out in batch mode, the length of the simulation depending on the complexity of the VHDL model. In the following paragraphs are described the results obtained when applying this injection strategy to the 8051 VHDL description while running two test bench applications.

4.3.2.1 ***Matrix multiplication:***



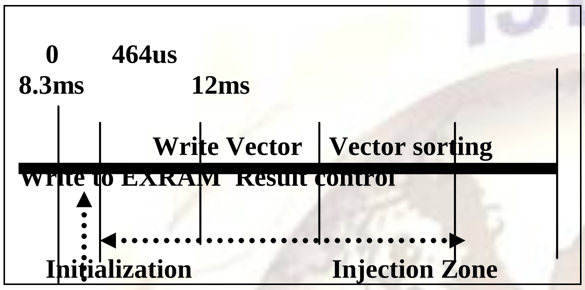
The matrix multiplication program operates in four phases, in the first phase, some internal registers are set in order to initialize the system, in the second phase, the 6x6 matrixes are generated and stored in the internal RAM, in the third phase the 6x6 product matrix is generated (this is the more time consuming phase), in the last phase, the result matrix is compared to the expected one and the incorrect results are stored in the external RAM. We define the “Test” as the union of all these four phases, and define the “Injection Zone” as the union of “generate matrix” and “product of matrix” phases.

At the end of each test the system provides the dump signal that generates a report of errors revealed during the result control phase.

The obtained results, in terms of percentages of errors with respect the total number of injected bit flips, are reported in the above table classified as tolerated errors, result errors and lost of sequences. “Tolerated errors”, correspond to those bit flips injected in memory elements which do not cause any effect at the outputs of the program. “Results errors”, gathers cases where obtained results are different from the expected ones. Finally, the cases where we do not get any answer from the processor are classified in the loss of sequence group.

The consequences of injected bit flips belonging to this last malfunction type are unrecoverable, needing to restart program execution. The results were analyzed in detail to determine the number of wrong elements in the result matrix obtained in each faulty execution.

An explanation of these results could be that the propagation of an injected SEU is related to the fault injection instant. In particular the experiments leading to six errors suggest that was corrupted a value of one of the 2 matrixes before they were multiplied. The latency of these unelaborated data inside the micro is quite long; in fact, if we inject the bit flip during the generation of the matrixes the incorrect value of the multiplicand matrix will generate 6 incorrect values on the result matrix.



4.3.2.2 Vector sorting program

In the first phase, some internal registers are set in order to initialize the system and a 30 element vector is generated and stored in the micro controller internal RAM. In the second phase, the original unsorted vector is stored in the external RAM. In the next phase the vector stored in the internal RAM of 8051 micro controller is sorted, this is the more time consuming phase. In the fourth phase the sorted vector is copied in the external RAM, this is the last phase of injection zone. In the last phase, the result of sorting algorithm is tested and the incorrect results are stored to the external RAM. Note that in this case the number of maximum possible errors is 30.

In this case we define the “Test” as the union of all these five phases, and define the “Injection Zone” as union of “Write vector”, “Vector sorting” and “Write to exRAM” phases. At the end of each test the system provides the dump signal that generates a report of errors revealed during the result control phase. The test follows the same steps of the previous one and the fault injection and result analysis are made in the same way. .

4.2.3. Use of built-in simulator commands

Two types of techniques based on the use of simulator commands can be identified: signal and variable manipulation. When using signal manipulation, faults are injected by altering the value of signals in the VHDL model. This is done by disconnecting a signal from its driver(s) and forcing it to a new value. The signal's driver(s) is (are) reconnected when the fault injection is completed. Variable manipulation allows injection of faults into behavioral models by altering values of variables defined in the VHDL code. The

Results of bit flip injection for the matrix multiplication program	**Values**
Injected Fault Numbers	**2416**
Lost of Sequences	**134**
Result Errors	**1068**
Tolerated Errors	**1348**
Result Errors Percentage (%)	**44.20%**
Lost of Sequences Percentage (%)	**5.54%**
Overall Percentage of Result Errors	**49.74%**

main reason for using built-in commands for fault injection is that this does not require a modification of the VHDL code. This technique is based on the use of the simulator commands to modify the value of the model signals and variables. The way that faults are injected depends on the injection place. To inject faults on signals, the following sequence of pseudo-commands must be performed:

1. Simulate until [injection instant]
2. Modify Signal [name] [faulty value]
3. Simulate for [fault duration]
4. Restore Signal [name]
5. Simulate for [observation time]

This sequence is thought to inject transient faults, which are the most common and difficult to detect . To inject permanent faults, the sequence is the same, but omitting steps 3 and 4. To inject intermittent faults, the sequence consists in repeating steps 1–5, with random separation intervals. The sequence of pseudo-commands to inject faults on variables is:

1. Simulate until [injection instant]
2. Assign Variable [variable name] [fault value]
3. Simulate for [observation time]

The operation is similar to the injection on signals, but in this case there is no control of the fault duration. This implies that it is not possible to inject permanent faults on variables using simulator commands. The sequence of commands needed to carry out the injection (for both transient and permanent faults) can be included in a macro file, where the elements between brackets will be passed to the macro as parameters. This means that the injection conditions can be varied without modifying the command code. It is interesting to point out that, from the point of view of the injection procedure, VHDL generics are managed as 'special' variables. This enables the injection of some nonusual fault types, such as delay faults (by modifying timing generics).

4.2.4. Comparison of fault injection techniques

The fault injection techniques presented above are compared in terms of fault modeling capacity, effort required for setting up an experiment and simulation time overhead. The implication of a fault injection campaign made up of a series of experiments is also considered. Mutants offer the highest fault modeling capacity of the fault injection techniques presented. They can be designed by using the full strength of the VHDL language itself, and are therefore well suited for implementing realistic behavioral fault models. They can be used also for implementing complex structural fault models, such as the stuck-open fault model for CMOS logic, in which a combinatorial circuit must be substituted for a sequential circuit. Saboteurs are generally less powerful than mutants in their fault modeling capacity. They can be used for simple fault models, such as the stuck-at fault model, but also for more complex fault models, for example by incorporating a finite state machine in the saboteur.

Signal manipulation is suited for implementing simple fault models, such as permanent or temporary stuck-at fault models. Variable manipulation offers a simple way for injecting behavioral faults. The usefulness of this technique is limited, however, as very few realistic behavioral fault models can be implemented by simply altering variable values. Both signal and variable manipulation can be used for controlling saboteurs and mutants. In this way, the injections of faults can be controlled by the built-in commands of the simulator when either mutants or saboteurs are used. If signal manipulation is used for this, the control signals should preferably be internal to the component description of the mutant/saboteur so that they do not have to be routed through the hierarchy of the VHDL model. The effort for setting up an experiment is small using signal and variable manipulation, as modification of the VHDL model is not required. More effort is needed for mutants and saboteurs, as they require:

(i) creation/generation of saboteurs/mutants,
(ii) Inclusion of saboteurs/mutants in the model and **(iii)** recompilation of the VHDL model.

Two ways can be distinguished for such an inclusion. One way is to generate a new configuration for each fault location, i.e., a configuration in which only one mutant is present at a time. This requires recompilation of the VHDL model for each fault location and may also require manual intervention to start up a simulation using the new model; both activities may incur a significant time overhead. Another way is to generate only one configuration in which all required mutants are included, and then activate these one at a time.

This may increase the simulation time depending on whether the mutants generate any supplementary events when fault injection is inhibited; thus, there is a trade-off between the overhead in simulation time and the overhead in compilation time. The creation of saboteurs and automatic generation of mutants is a relatively easy task, provided that simple fault models are considered. It is worth noting that a saboteur is a reusable component, while a mutant has to be specifically generated for each mutated component. The inclusion of saboteurs requires modification of the component description while mutants are easily included in the model by means of the VHDL configuration mechanism. The simulation time overhead imposed by signal and variable manipulation is only due to fault injection control, as the simulation must be stopped and started again for each fault injected. It is important to note that the simulation time overhead imposed by saboteurs and mutants depends on several factors: (i) amount of additional generated events (signal changes), e.g., a simple serial saboteur generates one event per input signal change, (ii) amount of code to execute per event, e.g., a complex behavioural mutant may require many statements to be executed per event, (iii) complexity of the fault injection control.

5. AUTOMATING THE INSERTION OF SABOTEURS:

A saboteur is a special VHDL component added to the original model. The mission of this component is to alter the value, or timing characteristics, of one or more signals when a fault is injected, remaining inactive during the normal operation of the system. In saboteurs are classified into: serial simple, serial complex and parallel. So far, VFIT can inject faults using serial saboteurs inserted manually in the design. The models of saboteurs implemented are as follows

5.1. Previous Models:

(**a**) Serial simple saboteur: It interrupts the connection between an output (driver) and its corresponding receptor (input), modifying the reception value.

(**b**) Serial simple bi-directional saboteur: It has two input/output signals, plus a read/write input that determines the perturbation direction.

(**c**) Serial complex saboteur: It interrupts the connection between two outputs and their corresponding receptors, modifying the reception values.

(**d**) <u>Serial complex bi-directional saboteur</u>: It has four input/output signals, plus a read/write input that determines the perturbation direction.

(**e**) <u>n-Bit unidirectional simple saboteur</u>: It is used in unidirectional buses of n bits (address and control). It is composed of n serial simple saboteurs.

(**f**) <u>n-Bit bi-directional simple saboteur</u>: It is used in bidirectional buses of n bits (data and control). It is composed of n bi-directional serial simple saboteurs.

(**g**) <u>n-Bit unidirectional complex saboteur</u>: It is used in unidirectional buses of n bits (address and control). It is composed of n=2 serial complex saboteurs.

(**h**) <u>n-Bit bi-directional complex saboteur</u>: It is used in bi-directional buses of n bits (data and control). It is composed of n=2 bi-directional complex saboteurs.

Faults can be injected on the signals which connect components in structural models. The internal architecture of the saboteurs can be behavioural or structural. The behavioural design is basically a process whose sensitivity list contains the control and input/output signals. The structural design is based on the use of multiplexers.

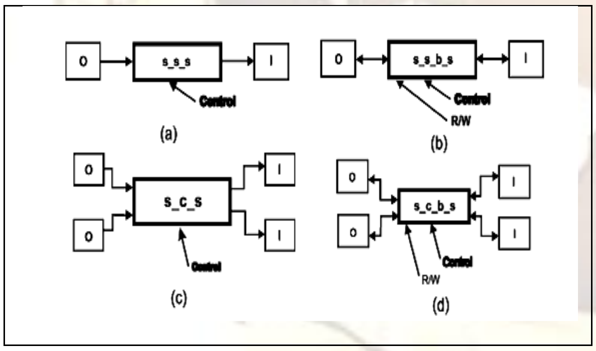


Fig5.1. (a) Serial simple saboteur (b) Serial simple bi-directional saboteur (c) Serial complex saboteur (d) Serial complex bi-directional saboteur

5.2. Fault Injection with Saboteur:

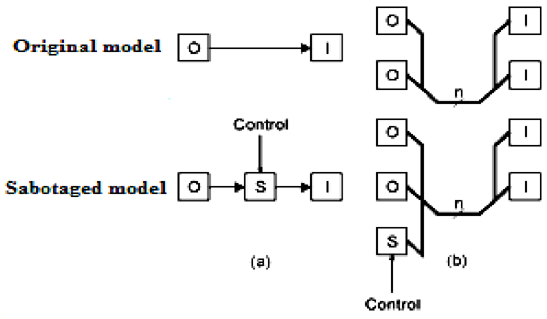


Fig: 5.1.1. (a) Serial
(b) Parallel saboteur Technique

A saboteur is a special VHDL component added to the original model. When activated, the mission of this component is to alter the value, or timing characteristics, of one or more signals, simulating the occurrence of a fault. During the normal operation of the system, instead, the component remains inactive. Saboteurs affect to the ports of the components in the model. Thus, this technique is applicable only to structural descriptions.

Attending to how saboteurs are inserted in the model, two types can be distinguished: serial and parallel. As Fig. 5.1.1 (a) shows, a serial saboteur interposes between a component input port (I in the figure) and its source signal (O in the figure), whereas a parallel saboteur Fig.5.1.1. (b) Is added as an additional source (S in the figure) of a given signal. Second, they allow to inject fewer fault models. For these reasons, their implementation has no special interest. So, in this paper, only serial saboteurs will be considered.

5.3 Enhanced models of saboteurs

This following new set of saboteur models has important differences with respect to prior ones.

• All models have been implemented using behavioural descriptions. This simplifies greatly their code and, what is more important, also the code of the design including the saboteurs. Moreover, the -bit versions can be used for vectors of any length, because their length is defined by means of a generic parameter. Every time an -bit saboteur is added to the model, the actual value of the generic parameter must be set.

• The number of saboteurs has been reduced to ease their automatic insertion. Now, depending on both the length (1 bit or n bits) and the mode (that is, the directionality) of the port sabotaged, only one model can be chosen.

• The bidirectional versions have the capability of injecting the fault only in the direction that data flow. In this way, the R/W input used in the models of prior version is not needed anymore, thus reducing the overhead. In the reduced version used to inject single faults, without the Control input, the spatial overhead is even more diminished.

• They can inject more fault models: pulse, short, and bridging the new models of saboteurs proposed, shown in Fig., are as follows.

• Unidirectional Serial Saboteur (USS): It is the same model as the SSS in the previous set, although the USS allows injecting new fault models.

• Bidirectional Serial Saboteur (BSS): It is similar to the SSBS in the first set, but like in the previous case, the fault model set that can be injected has been extended. Also, it eliminates the R/W control signal.

•N-Bit Unidirectional Serial Saboteur (NUSS): This model replaces all the unidirectional multi-bit models in prior model set.

• N-Bit Bidirectional Serial Saboteur (NBSS): It replaces all bidirectional multi-bit models in the former proposal and eliminates the R/W control signal.

As the timing of Control and Selection inputs are identical, we have implemented an "optimized" version of these models in which the fault injection is managed only by Selection input.

The idea is simple:

When an injection is in progress, Selection indicates the fault(s) to be injected; but while no fault is injected, the value of Selection must represent a "no-fault" injection.

However, this reduced version has a negative aspect: only single faults and multiple faults in the domain of time can be injected. To inject faults in the domain of space, the original scheme must be used.

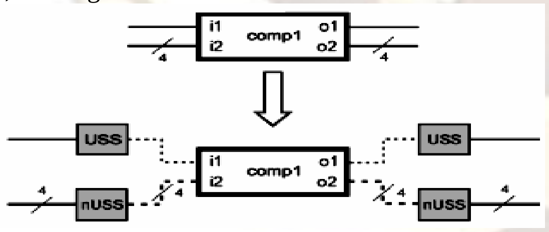


Fig5.2.1 Example of perturbation of model Distribution of saboteur

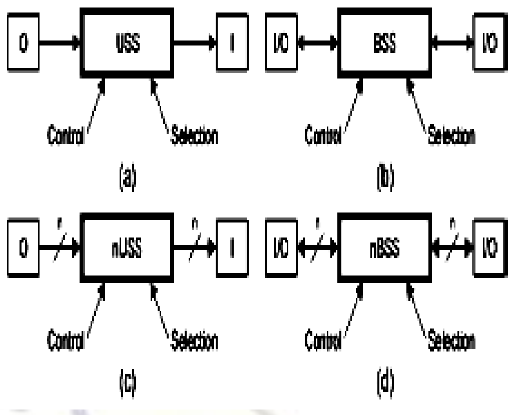
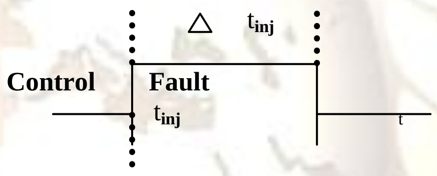


Fig5.2.2. Set of Saboteur implemented

(a) Unidirectional serial saboteur

(b) Bidirectional saboteurs

(c) n-USS (d) n-BSS



Every saboteur is controlled by means of the following three inputs.

• **Control:** whose mission is the timing of the injection: its activation determines both the injection instant (**t_{inj}**) and the fault duration (**Δt_{inj}**). It can be seen more clearly in the above figure.

• **Selection:** this allows selecting the fault model to be injected.

• **R/W:** this indicates, in the bidirectional versions, the direction of the perturbation.

The task of modifying automatically a source code seems apparently very complex. However, if the injection tool includes a parser, this is not actually so. From a syntactical tree of the model containing its complete structure, it is possible to go over the tree and generate a new copy of the source files, inserting new sentences or modifying other existing as needed.

The insertion of saboteurs involves the following three actions:
1) Declaring the signals required to activate the saboteurs and to select the fault model to be injected;

2) Declaring the components of the saboteurs introduced;
3) Inserting the instances of the saboteurs, interposing between local and formal ports of the sabotaged components; this also implies declaring new signals to connect the saboteurs to local ports, and modifying the original mapping of ports.

5.4. Fault Injection Using Mutants

A mutant is a component that replaces another component. While inactive, it works like the original component, but when it is activated, it behaves like the component in presence of faults.

The mutation can be made in three ways:

• By adding saboteurs to structural model descriptions.

• By modifying structural descriptions replacing sub-components.

• By modifying syntactical structures of behavioral descriptions.

There can exists lots of possible mutations in a VHDL model, so representative subsets of faults at logical and RT levels must be considered replacing the values of conditions in if and case statements (called stuck-then, stuck-else, dead clause, etc.), disturbing assignment statements (assignment control, global stuck-data, etc.), disturbing operators in expressions (micro-operation, local stuck-data), etc.

Here we considered the following fault models

• **Stuck-Then**: Replacement of the 'if' condition by true.
• **Stuck-Else**: Replacement of the 'if' condition by false.
• **Assignment Control**: Disturbing an assignment operation.
• **Dead Process**: Elimination of the sensitivity list of a process.
• **Dead Clause**: Elimination of a clause in a case.
• **Micro-Operation**: Disturbing an operator.
• Local Stuck-Data: Disturbing the value of a variable, constant, or signal in an expression.
• **Global Stuck-Data**: Elimination of all value modifications of a variable or signal in architecture.

Many of these fault models do not have a direct correspondence with physical faults, but they can show somehow an erroneous internal operation.

5.4.1 Graphical User Interface

The experiment configuration is carried out through VFIT's graphic user interface (GUI). Among other functions, this GUI allows the user to select a list of fault targets among all the possible targets in the model. The class of the fault targets eligible depends directly on the fault injection technique applied like model signals and variables for simulator commands; inputs and internal connection signals of the model components for saboteurs; and special VHDL sentences for mutants.

5.4.2 Injection Schedulers

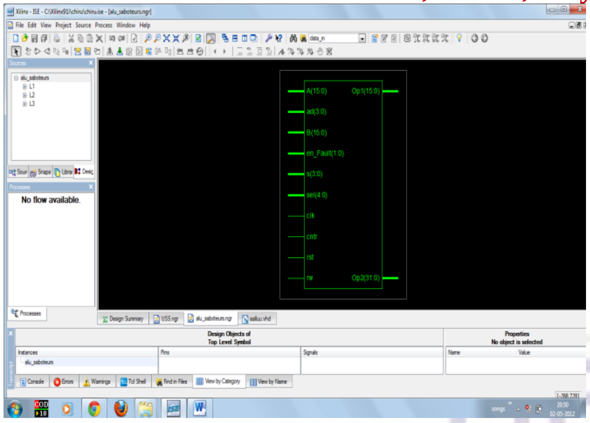
It decides at a given time instant (injection instant), the value of one or several points of the system must behave in a wrong way. If it occurs once for a short time then it is simulating the occurrence of a transient fault, if it occurs for a short time but repeatedly then it is simulating the occurrence of an intermittent fault or permanently i.e. until the end of the simulation.

5.4.3 Dependency

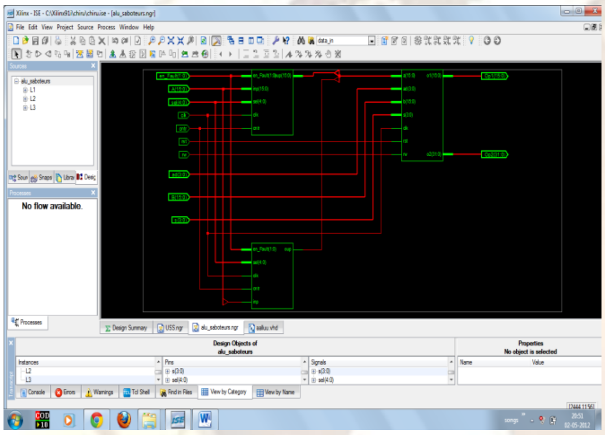
Wrong behavior of the fault targets means depends strongly on the injection technique used. In the case of simulator commands, the injection consists on modifying directly the internal value or timing of the fault target(s) by using the commands of a simulation language. When saboteurs are used, the injection consists on modifying directly the control lines that manage one or several saboteurs inserted in the original model. In this way, the saboteur(s) activated will propagate the affected lines with erroneous values or timing. When injecting faults with mutants, the, By means of simulator commands, an erroneous sentence will be "executed" instead of the correct one. During the simulation phase, VFIT automatically selects randomly a fault target from the list, and then, a particular fault model to inject on it. The output of an injection experiment can be either an error syndrome analysis or a validation.

6. SIMULATION RESULTS

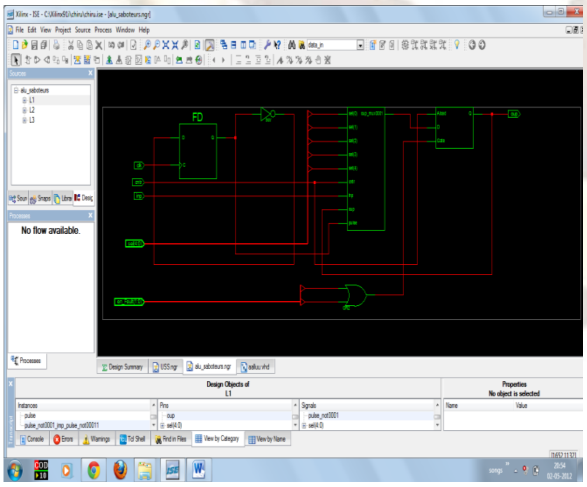
6.1. Saboteurs Top-level



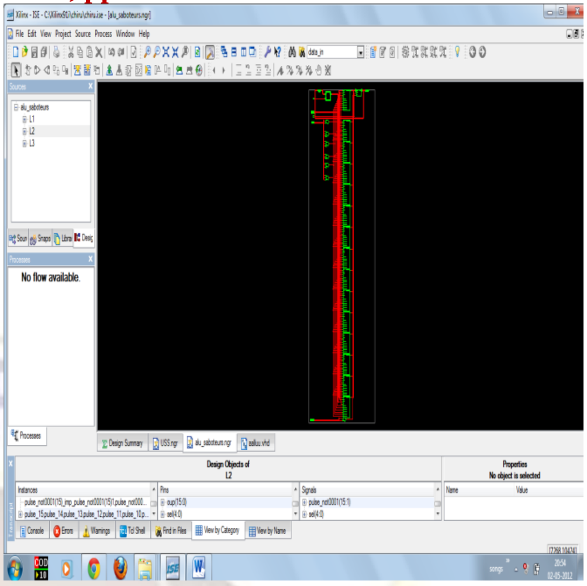
6.2. Saboteurs Top-level Internal block diagram



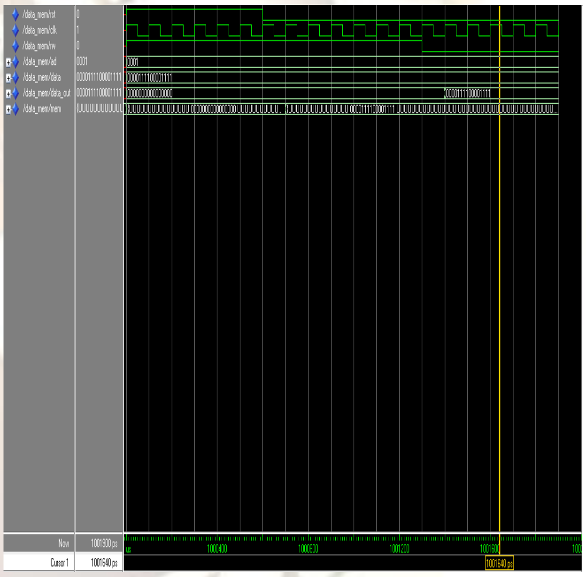
6.3. USS:



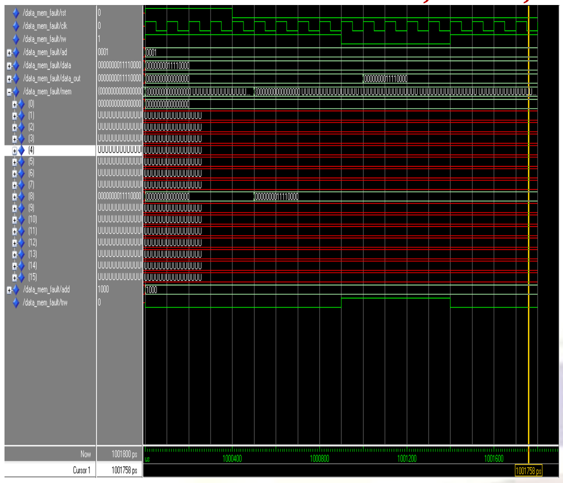
6.5. USS 16:



6.6. MEMORY output without fault application



6.7. MEMORY output with fault application



7. CONCLUSION

In this project, we introduced new methods to implement and use saboteurs and mutants into VHDL models. The new models of saboteurs fix some problems of ambiguity that the previous approach had. These problems prevented their automatic insertion. Moreover, the new models have been implemented in such a way that they diminish the overhead, by reducing the number of signals required to manage bidirectional saboteurs. Another enhancement respect to prior models is that they allow injecting more fault models. The advantages of the new proposal to implement mutants are especially relevant: it is easy to automate and reduces notably the spatial overhead. But its main success is to shrink considerably the temporal overhead. These saboteurs and mutants have been applied to example circuits at gate level and register level. Thus their output waveforms have been observed using Simulation by Model Sim-6.2g.

Acknowledgements

The authors like to express their thanks to management and department of ECE KLUniversity for their support and encouragement during this work.

8. References:

1. E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson, “Fault injection into VHDL models: The MEFISTO tool,” in Proc. FTCS, 1994, pp. 356–363

2. V. Sieh, O. Tschäche, and F. Balbach, “VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions,” in Proc. FTCS, 1997, pp. 32–36.

3. J. Boué, P. Pétillon, and Y. Crouzet, “MEFISTO-L: A VHDL-based fault injection tool for the experimental assessment of fault tolerance,”

4. J. Arlat, Y. Crouzet, and J.C. Laprie, “Fault Injection for Dependability Validation of Fault-Tolerant Computer Systems,” Proc. 19th Ann. Int’l Symp. Fault-Tolerant Computing, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 348-355.

5. O. Gunnetlo, J. Karlsson, and J. Tonn, “Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation,” Proc. 19th Ann. Int’l Symp. Fault-Tolerant Computing, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 340-347.

6. C. Constantinescu, “Impact of deep submicron technology on dependability of VLSI circuits,” in Proc. DSN, 2002, pp. 205–209.

7. P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, “Modeling the effect of technology trends on soft error rate of combinational logic,” in Proc. DSN, 2002, pp. 389–398.

8. J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell, “Fault injection for dependability validation: A methodology and some applications,” IEEE Trans. Softw. Eng., vol. 16, no. 2, pp. 166–182, Feb. 1990.

9. Fault Injection Techniques and Tools for VLSI Reliability Evaluation, A. Benso and P. Prinetto, Eds. Norwell, MA: Kluwer Academic, 2003.fault injection tool for the experimental assessment of fault tolerance,” in Proc. FTCS, 1998, pp. 168–173.

10. Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code Juan-Carlos Baraza, Joaquín Gracia, Sara Blanc, Daniel Gil, and Pedro-J. Gil, Member, IEEE

11. C. Constantinescu, “Impact of deep submicron technology on dependability of VLSI circuits,” in Proc. DSN, 2002, pp. 205–209.

Susrutha Babu Sukhavasi was born in India, A.P. He received the **B.Tech** degree from JNTU, A.P, and **M.Tech** degree from SRM University, Chennai, Tamil Nadu, India in 2008 and 2010 respectively. He worked as **Assistant Professor** in Electronics & Communications Engineering in Bapatla

Engineering College for academic year 2010-2011 and from 2011 to till date working in **K L University**. He is a member of Indian Society For Technical Education and International Association of Engineers. His research interests include Mixed and Analog VLSI Design, FPGA Implementation, Low Power Design and wireless Communications, Digital VLSI. He published articles in various international journals and conference.

Suparshya Babu Sukhavasi was born in India, A.P. He received the **B.Tech** degree from JNTU, A.P, and **M.Tech** degree from SRM University, Chennai, Tamil Nadu, and India in 2008 and 2010 respectively. He worked as **Assistant Professor** in Electronics & Communications Engineering in Bapatla Engineering College for academic year 2010-2011 and from 2011 to till date working in **K L University**. He is a member of Indian Society For Technical Education and International Association of Engineers. His research interests include Mixed and Analog VLSI Design, FPGA Implementation, Low Power Design and Wireless communications, VLSI in Robotics. He published articles in various international journals and conference.

Dr.Habibulla khan born in India, 1962. He obtained his B.E. from V R Siddhartha Engineering College, Vijayawada during 1980-84. M.E from C.I.T, Coimbatore during 1985-87 and PhD from Andhra University in the area of antennas in the year 2007.He is having more than 20 years of teaching experience and having more than 20 international, national journals/conference papers in his credit.Prof. Habibulla khan presently working as **Head of the ECE department at K L University**. He is a fellow of I.E.T.E, Member IE and other bodies like ISTE. His research interested areas includes Antenna system designing, microwave engineering, Electro magnetics and RF system designing.

Chiranjeevi Pilla was born in garividi, vizianagaram(dist), a.p, india. He received the B.Tech. degree in Electronics & Communications Engineering from St. Theressa inistitute of Engineering &Technology, garividi, A.P., Affiliated to the JNTU , Kakinada, A.P., India in 2009 and pursuing M.Tech Degree in VLSI technology in KL University. His research interests include Digital VLSI Design and Fault Diagnosis & testing and Verification.