RASP-FIT: A Fast and Automatic Fault Injection Tool for Code-Modification of FPGA Designs

Abdul Rafay Khatri, Ali Hayek, and Josef Börcsök
Department of Computer Architecture and System Programming
University of Kassel
Kassel, Germany

***Abstract*—Fault Injection (FI) is the most popular technique used in the evaluation of fault effects and the dependability of a design. Fault Simulation/Emulation (S/E) is involved in several applications such as test data generation, test set evaluation, circuit testability, fault detection & diagnosis, and many others. These applications require a faulty module of the original design for fault injection testing. Currently, Hardware Description Languages (HDL) are involved in improving methodologies related to the digital system testing for Field Programmable Gate Array (FPGA). Designers can perform advanced testing and fault S/E methods directly on HDL. To modify the HDL design, it is very cumbersome and time-consuming task. Therefore, a fault injection tool (RASP-FIT) is developed and presented, which consists of code-modifier, fault injection control unit and result analyser. However, in this paper, code modification techniques of RASP-FIT are explained for the Verilog code at different abstraction levels. By code-modification, it means that a faulty module of the original design is generated which includes different permanent and transient faults at every possible location. The RASP-FIT tool is an automatic and fast tool which does not require much user intervention. To validate these claims, various faulty modules for different benchmark designs are generated and presented.**

***Keywords*—Code generator; Fault emulation; Fault injection; Fault simulation; Instrumentation; Parser**

I. Introduction

Hardware Description Languages (HDL) have been involved in improving various methodologies related to digital system testing during the last few decades. This reduces the gap between the tools and methodologies used by design and test engineers. Using HDL, the design engineers can verify and test the design at an early stage, and there is no need to convert the design into a compatible format [1]. Verilog HDL is one of the most widely used languages for implementing the design structure for Application Specific Integrated Circuit (ASIC) and FPGA-based designs [2]. These designs are mostly written in HDL and a bit-stream file is generated, which is downloaded into the FPGA chip to implement the design. The FPGA development flow consists of various processes, e.g. synthesis, translate, place & route, and then a bit-stream generation. Various fault injection tools have been devised in the past several years for FPGA-based designs, which work on different stages of the development flow [3], [4] as shown in Fig. 1. It depicts the way of injecting faults at various stages of FPGA development flow.

Generally, FI techniques are divided into four: namely hardware, software, simulation, and emulation-based. Particularly, for FPGA-based systems, emulation and simulation-based techniques are involved in testing, dependability analysis and fault simulation/emulation applications [5]. Designs implemented on the FPGA are also prone to errors and failures, due to radiations and several other reasons, so it is necessary to test and verify the designs. Both testing and verification involve a deliberate introduction of faults in the System Under Test (SUT). Fault injection technique is used in the process of evaluation of fault effects and fault tolerance [6]. The fault injection technique consists of the deliberate insertion of faults into the particular target system and monitors the responses to observe the effects of the faults. In a nutshell, the fault injection technique provides:

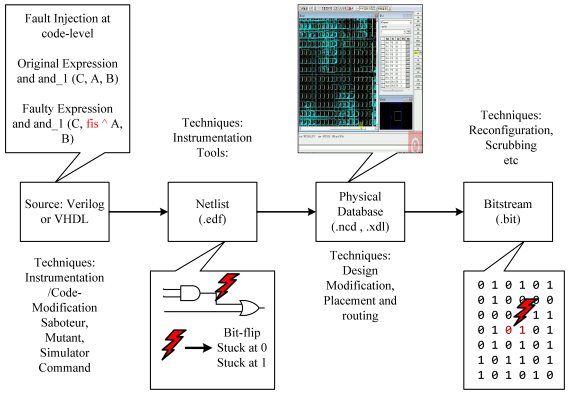


Fig. 1. Fault injection techniques at various stages of the FPGA development flow.

- Statistical estimation of soft-errors for dependability analysis.
- Evaluation of design characteristics for reliability.
- Measurement of the effectiveness of fault tolerance capability of design.
- Ability to find the critical components of an overall design.
- The way to test the digital design and obtains the test vectors for the automatic test equipment.
- Fault coverage and code coverage for the design in the verification process.

There are several reasons for involving FPGA in developing of fault injection techniques and tools, such as prototype availability of designs (for simulation), fast emulation (also the high speed of injections), more on-chip area availability and

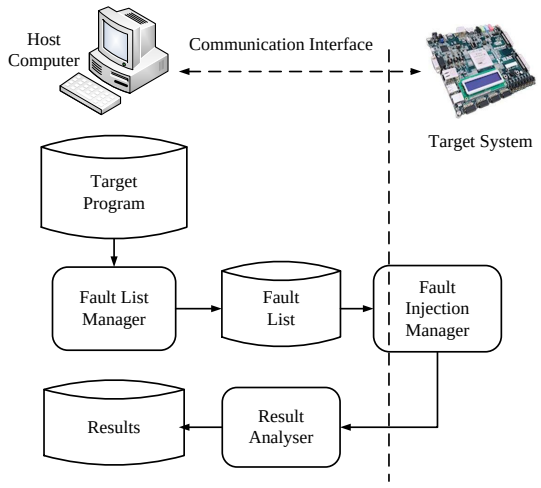


Fig. 2. Fault injection environment [7].

(full & partial) reconfiguration techniques. The main issue in developing a fault injection tool is describing the mechanism to inject, select, and activate a particular fault. In general, any fault injection tool consists of these three basic building blocks such as fault list manager, fault injection manager, and a result analyser as shown in Fig. 2. FI tools for FPGA designs are classified into two main categories and divided further as shown in Fig. 3.

The RASP-FIT (RechnerArchitektur and SystemProgrammierung)–German name of the institute– Fault Injection Tool is presented in this work, which consists of three main parts such as Fault Injection Algorithm (FIA), fault injection control unit, and result analyser [5]. In this paper, the FIA is focused which takes synthesizable Verilog file as an input, parses the code, finds the locations and instruments/modifies the file to generate the faulty design to perform the fault injection and fault simulation/emulation analysis of ASIC and FPGA-based designs. The tool, with its graphical user interface, is developed in Matlab. This fault injection tool deals with various fault models (e.g. bit-flip & stuck-at 1/0) and able to generate any number of faulty designs (required by the user) of the original design with evenly distributed faults in them. It adds the proposed fault control unit (e.g. the FISA Unit) with the required

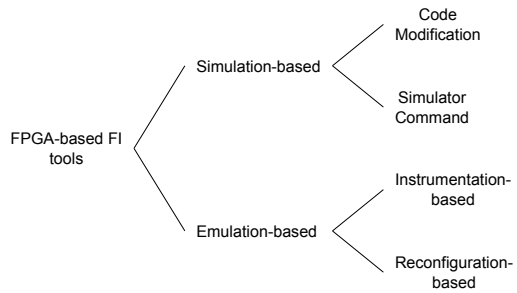


Fig. 3. FPGA-based fault injection techniques and tools.

ports in the faulty HDL design. The RASP-FIT tool is fast and user-friendly and it takes an appropriate time for the generation of faulty modules of the original design. Various benchmark circuits are considered and their compilable faulty modules are generated and presented. The complexity of a design, the way of injecting faults at each code abstraction level and the total number of faults injected in the design versus time taken by this tool is evaluated and presented in this work.

This paper is organized as follows: Section II presents some background information. An automatic Verilog code modifier tool is presented in Section III. Section IV presents the description of the RASP-FIT tool developed in Matlab. Section V shows the results of the instrumentation of Verilog design code at various abstraction levels with timing analysis. Section VI concludes the paper.

II. Background

The ongoing miniaturisation of digital systems makes them more and more sensitive to faults, which complicates the design process of fault-tolerant systems. In this situation, fault injection plays an important role in the process of testing, verifying system's robustness and fault-tolerance capabilities. In the last few years, the fault injection technique is directly applied to the FPGA-based designs, written in HDLs, mainly Verilog or VHDL. Using HDL, designers can use different existing test methods and develop new ones with little effort [1]. Fault injection techniques and tools for FPGA-based designs are divided into two major categories in the literature.

A. Simulation-based Fault Injection Tool for FPGA

Simulation-Based Fault Injection (SBFI) techniques can be categorized into two, i.e. Code Modification (CM) and Simulator Command (SC). The first technique requires modification of HDL code by adding saboteurs and mutants, whereas, the signal or variable values are changed through simulator commands in the second technique [8]. The advantages of using SBFI technique are [9], [10]:

- There is no risk of damage of the SUT.
- Cost effective because no real hardware is used.
- Higher observability and controllability during fault injection campaigns.
- Modelling of both transient and permanent faults is achieved with ease.
- Supports all abstraction levels.

At the code level, the fault injection techniques for the FPGA designs usually come in the category of SBFI. There are many tools that are designed and available for VHDL in the literature, e.g. VERIFY [11], (MEFISTO-C, HEARTLESS, VFIT, FTI) [12], [9], FSFI [13] etc. All these tools are developed for VHDL based designs using SC, saboteur and mutant techniques. The application of Verilog PLI includes test generation during fault simulation. This environment is capable of fault injection, generation of some random patterns and check the responses of injected faults [1]. In some approaches, top-level design module is modified, along with the simulator command technique as presented in [23].

B. Emulation-based Fault Injection Tool for FPGA

The FPGA design & development flow consists of many stages, where modification of the design is possible for the fault injection analysis. Emulation-based fault injection techniques can be categorized into two, i.e. instrumentation and reconfiguration. The advantages of using emulation based fault injection technique are [9], [12]:

- Injection time can be improved as compared to SBFI
- Time and area overhead reduction using partial reconfiguration technique
- Higher observability and controllability

Authors studied the recently developed fault injection tools based on instrumentation technique in the FPGA development flow, such as, tools that work on the net-list developed after the synthesis process [14], [15], [16], some tools based on the instrumentation technique on the code level [18], [17], and using some hybrid techniques (simulation/emulation) [19], [20], [21], [22]. HDL environment is able to generate a list of faults and it is used for fault emulation/simulation of the target system. Authors in [24] presented a code modifier which is developed in C++ language for structural Verilog net-list. A multiplexer is injected as a stuck-at fault model in the code. In comparison with this work, the RASP-FIT tool injects bit-flip and stuck-at (1/0) fault models using simple gates XOR, OR, AND with NOT, respectively. Hence, the RASP-FIT tool provides a small number of additional input ports and area overhead.

The main goal is to develop a fault injection tool, which performs fault injection analysis, fault simulation/emulation, testing, and dependability analysis directly on HDL designs for FPGAs and ASICs. This can reduce the gap between the tools and methodologies used by design and test engineers which speed-up the process of testing, produce cost-effective methods and reduce the time to market. In this paper, code-modification techniques of RASP-FIT tool for various abstraction levels are presented in detail.

III. Automatic Code Generator (A Verilog Code-Modifier)

The concept of automatic code generation involves a number of various techniques such as code completion or code insertion. The code transformation is a technique, where a piece of code is transformed into a target language from a source code [25]. In this work, an automatic code generator is developed to generate a compilable faulty module of the original design, written at Verilog HDL. These faulty designs can be used for fault S/E analysis and testing of FPGA-based designs. The automatic code modifier serves following basic functions:

1) Reading of design file (code parsing).
2) Instrumentation of design code and generation of faulty design code.
3) Addition of fault control unit in each faulty module.
4) Writing of instrumented/modified code to a file having *.v extension.

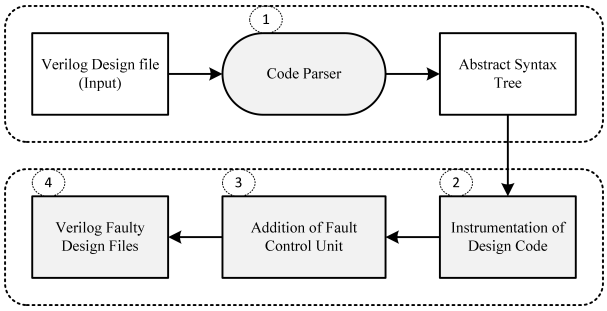


Fig. 4. Block diagram of the automatic code generation process.

The block diagram summarizes these functions, developed under RASP-FIT tool as a Verilog code modifier is shown in Fig. 4.

A. Code Parsing Technique in RASP-FIT

The code parser is a fundamental component of the RASP-FIT tool, which analyses the design code written in Verilog HDL. Normally, a parser generates an Abstract Syntax Tree (AST) from the design code for further analysis. As described earlier, the Verilog code for FPGA-based designs is written at various abstraction levels. The automatic code modifier developed under RASP-FIT is able to modify/instrument the design at any abstraction level for fault injection analysis. The interpretation of the developed parser technique in RASP-FIT for fault injection is shown in Fig. 5 for a gate level design. For different abstraction levels, the way of injecting faults in the design is also different. Detailed about each level is given in the sequel.

1) Gate-level Designs: At gate abstraction level, the basic cell of the design is a logic gate. A logic circuit which contains a few hundreds of logic gates are typically designed at this level [4].

TABLE I. Pre-Defined Gate Primitives in Verilog HDL

S. No.	Gate primitives	I/O positions
1	and, or, nor, nand xor, xnor	First terminal is output, one or more inputs
2	buf, not	One or more outputs, last terminal is input
3	bufif0, bufif1, notif0, notif1	First terminal is output, Second terminal is input, Third terminal is control

Gate level coding of any design in Verilog HDL consists of built-in gate primitives e.g.(`and, or, nand, nor, xor, bufif0, etc.`), and user-defined primitives. In these primitives, some ports are assigned as outputs and some as inputs. Their positions are defined in Verilog HDL. Table I shows a review of built-in primitives with their positions of inputs/outputs. By default, the RASP-FIT tool injects faults at the input positions, whereas, these positions can also be defined in the library for user-defined primitives. To inject faults at output ports, the RASP-FIT tool adds buffer (buf) to each port. The way of fault modification at this level for the bit-flip fault model is shown in Fig. 6. In this figure, `f0,f1` represents the bit-flip faults in this line of the code.

```
// Verilog
// c17 benchmark circuit ISCAS'85
// Ninputs 5
// Noutputs 2
// NtotalGates 6
// NAND2 6

module c17 (N1,N2,N3,N6,N7,N22,N23);

input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;

nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);

endmodule

'//' Comments (single or multi-line '/*
    */') are ignored and removed in faulty
    copies of original design.

ModuleName: c17, (used in generation of
    other copies names as c17_faultycopy1,
    c17_faultycopy2, ... etc.)

Input Port List : Keep it in the container.
    Map() with their dimensions (if vector)
    .
Output Port Declaration: Output port's name
     are changed with the inclusion of
    outVar_f1 or outVar_f2 etc for further
    comparison in fault injection.
Wire Declaration: Keep it in the container.
    Map() with their dimensions (if vector)
    .
nand : Recognise gate_level design (Gate-
    level Library added in the tool
    contains prototypes of built-in and
    user defined primitives).
Consturct fault list & count fault
    locations : 12
End of code
```

Fig. 5. Parsing of a Verilog design file.

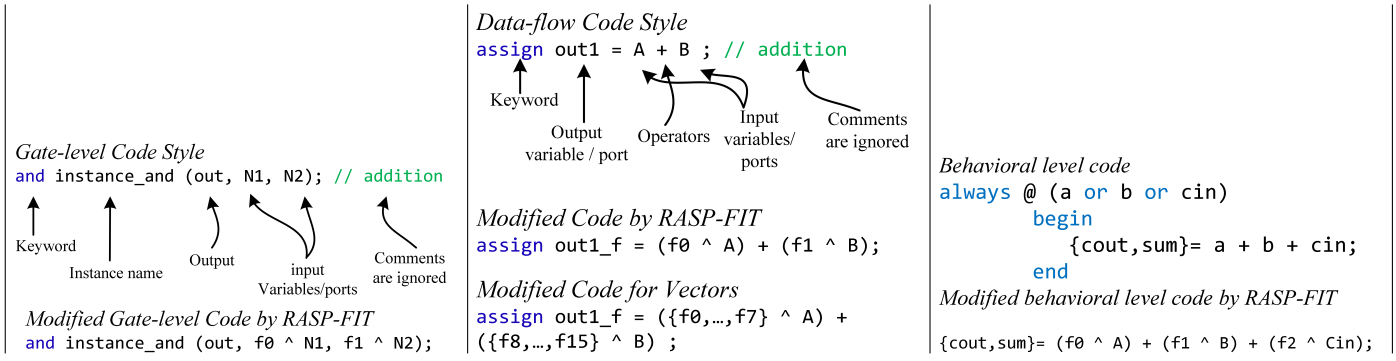


Fig. 6. Fault injection style for different abstraction level.

TABLE II. VERILOG OPERATORS ADDED IN RASP-FIT FOR DATA-FLOW ABSTRACTION LEVEL

Operators (Op) used in a Data-flow Abstraction Level	Original Expression	Faulty Expression by RASP-FIT Tool	Precedence
Unary Operators			
Unary (+, -)	assign B = -A;	assign B = fn ^ -A;	Highest
∼, & , — , ∼&, ∼—, ^, ∼ ^,	assign B = OpA	assign B = fn ^ OpA;	
Binary Operators			
Arithmetic (+, -, *, /, %, **)	assign C = operand1 Op operand2;	assign C = (fn ^ operand1) Op (fn+1 ^ operand2);	
Relational (<, >, <=, >=)	–	–	
Equality (= = =, != =, = =, !=)	–	–	
Logical (&&, — —)	–	–	
Ternary Operator			
Conditional (? :)	assign C = expr1 ? expr2 : expr3;	assign C = expr1 ? (fn ^ expr2) : (fn+1 ^ expr3);	Lowest

2) Data-flow Designs: For small circuits, up to a few thousand of gates, the gate level modelling approach can work very well. However, the large circuits consist of hundreds of thousands of gates and gate-level modelling is not feasible to test and verify the circuit. Data-flow modelling provides a powerful way to implement large and complex designs. Data-flow is a bit higher level of abstraction than gate-level modelling.

The `assign` command is the heart of Verilog data-flow abstraction level [26], [27]. Fig. 6 shows the fault injection logic for data-flow designs. A simple expression is shown with `assign` statement. For a single bit variables, only one fault is injected per variable. Similarly, in the second modified expression, these variables are considered a byte long variables. Therefore, with the help of concatenation and bit-wise reduction operators, we can inject faults in the vectors. Note that, integer variables (`integer`) are considered 32-bit wide, for the code modifier developed under RASP-FIT tool. It reads and stores the declaration variables with their lengths for fault injections. Table II describes the summary of most widely used operators in data-flow designs with the examples of correct and faulty expressions for bit-flip fault model.

3) Behavioural Designs: Modelling a circuit with logic gates and continuous assignments reflects quite closely the logic structure of the circuit being modelled; however, these constructs fail to describe complex high-level aspects of a system [28]. Verilog provides designers with the ability to describe the whole design functionality in an algorithmic manner, which represents the behaviour of the design [27]. Verilog's behavioural construct is similar to C language construct and it provides greater flexibility to designers.

Verilog behavioural models contain procedural statements that control the simulation and manipulate variables of the data types. The major components of behavioural constructs consist of: always and initial blocks, blocking and non-blocking assignments, conditional statements, multi-way branching, looping statements, sequential and parallel blocks etc. Note that, the vectors are treated with the same approach as described in a data-flow abstraction level. Prototypes for each expression and operators are added to the code modifier. When the code modifier reads the line of code and extracts the command (keyword), it injects the fault accordingly. Faults are injected into the right-hand side of the expression as shown in Fig. 6.

B. Instrumentation Technique for Verilog HDL

The instrumentation is a technique in which extra circuitry added to the design for fault injection/simulation applications, which is commonly known as 'saboteur'. In normal operation, it remains inactive, but when it is activated, it injects faults in the SUT during the fault injection process. The benefit of using this technique is that it does not have time limitations during circuit operation. In FPGA development flow, instrumentation of additional circuits can be done at various stages, e.g. in net-list, bit-stream, and HDL design code. In the RASP-FIT tool, `(XOR, OR, AND with NOT)` gates are used to inject bit-flip, and stuck at 1/0 faults respectively.

1) Fault Models in Verilog HDL: The fault models are developed to be used in pretending the defects in the test process and dependability analysis. Faults can be classified into various categories, such as permanent, transient and intermittent faults. In simple words, a fault is a manifestation of error [29]. Some fault models are widely used in digital circuit testing, fault simulation/emulation, and dependability analysis. These fault models are stuck at fault and Single Event Upset–commonly known as bit-flip–(SEU) fault. The fault injection technique at code level should describe the way to inject these faults in the code, which pretend as real faults occurred in the system, given in the sequel.

Stuck-at Fault Model in Verilog HDL: The stuck-at fault is a fault on a line or its interconnecting gates, which causes the logic value to be appeared on the line never changes. There are two categories of such fault model, i.e. stuck at 1 (sa-1) and stuck at 0 (sa-0) [1], [30]. In the sa-1 fault model, a logic value '1' appears to a signal line in the logic circuit, whereas, in the sa-0 fault model a logic value '0' appears on a line. Two faults per line can occur, these are sa-1 or sa-0 at the input or the output of a logic gate. In Verilog HDL, these faults can be injected into the gate, data-flow, and behavioural abstraction levels as shown in Fig. 6.

Bit-flip Fault Model in Verilog HDL: The bit-flip fault model is also widely used in order to calculate SEU. An SEU occurs when a bit is changed from logic '0' to logic '1' and vice versa.

Table III presents the summary of all fault models along with the Verilog operators used in RASP-FIT code modifier.

fn and Var show the particular fault and the declared input, wire or reg ports in the design, respectively.

TABLE III. SUMMARY FOR FAULT MODELS AND VERILOG OPERATORS

S.No.	Fault model	Verilog operator	Verilog code
1	Stuck-at 1	| (OR logic)	(fn | Var)
2	Stuck-at 0	∼,& (AND logic)	(∼fn & Var)
3	Bit-flip (SEU)	^(XOR logic)	(fn ^Var)

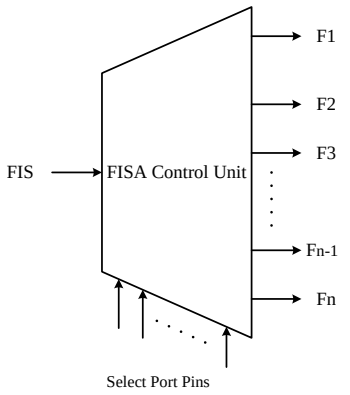
C. Fault Control Unit: FISA Unit

DEMUX-based Fault Injection, Selection, & Activation (FISA) unit is developed to control the selection and activation of the injected faults in the fault simulation/emulation applications as shown in Fig. 7. The proposed fault control unit is a simple unit and it provides the high controllability and observability about the selection and activation of faults. The FIS signal has a logic value '1'. When the select port is assigned a value from a test-bench (in simulation), then that fault is selected and activated in the target system. The fault injection analysis under RASP-FIT includes the code for the FISA unit in each faulty copy of the target design [3]. As we are generating a faulty module of the original design at the code level, we need to write this code in the design. It is very time-consuming to write HDL code manually for the fault control unit when a number of faults are large enough. For that purpose, a function (gen_always) is created and integrated into the RASP-FIT tool. The output of the function is shown in Fig. 7 and it is added to the faulty design as shown in Fig. 9.

IV. DEVELOPMENT OF RASP-FIT IN MATLAB

The RASP-FIT tool is developed in Matlab along with its graphical user interface. The tool consists of three major functions, namely, fault_injection(), static_compaction() and hardness_analysis(). All these functions are developed in Matlab under the function RASP_FIT(). It is a tabbed-based GUI as shown in Fig. 8. Each tab performs certain specific functions.

In this paper, the fault injection capability of this tool is presented. The fault_injection() function consists of approximately 563 lines of code having 20 functions. A Verilog code modification is described in detail in Section V. At the graphical user interface, the user must provide a synthesizable Verilog design file as an input, select the type of fault model for injection in the design from a drop-down menu and enter the number of faulty modules required. By clicking on the Generate button, faulty modules will be generated along with the top file. The faulty modules are saved under the name (moduleName_faultycopy1.v, moduleName_faultycopy2.v and so on) at the same location/folder. The top file, which contains the comparator logic and memory declaration for storing results of the comparisons, is saved under the name (moduleName_top.v). These modified designs are now used for the fault simulation/emulation, digital testing and dependability analysis, with FPGA tools, without much effort.




```
wire fis = '1'; // Declaration part
reg f0,f1,f2;

always @(select) begin
  if (select == 2'd0) begin
    f0=fis;f1=0;f2=0; end
  else if (select == 2'd1) begin
    f0=0;f1=fis;f2=0; end
  else if (select == 2'd2) begin
    f0=0;f1=0;f2=fis; end
  else begin
    f0=0;f1=0;f2=0; end
  end
```

Fig. 7. Proposed DEMUX-based FISA control unit (above), and it's Verilog code (below).

V. RESULT AND DISCUSSION

The RASP-FIT tool has the capability to modify (instrument) the Verilog code, written at any abstraction level. As described earlier, there are three main abstraction levels, e.g. gate-level, data-flow, and behavioural levels. The fault injection technique is widely used in fault simulation/emulation, digital testing and dependability analysis. To perform fault injection, we need a faulty module of an original module (i.e. golden module). In our case, the golden module is available in Verilog code for FPGA-based designs. In order to generate a compilable faulty code of the original design with the inclusion of faults and fault control unit, we need to modify the code which is a cumbersome and time-consuming task. When the complexity of design is increased, it injects more faults and takes more time to generate faulty copies.

In this work, we have used different benchmark circuits (ISCAS'85, ISCAS'89, EPFL and some behavioural designs). These benchmark circuits are written in gate-level, data-flow and behavioural abstraction levels. The complexity of design in terms of logic gates and time taken for these design is described tabularly. Details are given in the sequel.

A. Gate Abstraction Level Code

To validate a test methodology, the ISCAS'85 and ISCAS'89 benchmark circuits are most widely used. These benchmark circuits consist of combinational and sequential circuits. The ISCAS'85 consists of 11 combinational benchmark circuits, whereas the ISCAS'89 consists of 23 sequential

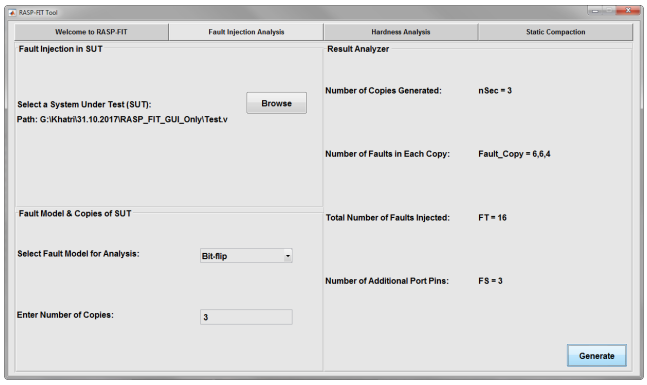


Fig. 8. Tabbed-based GUI of the proposed tool (RASP-FIT).

circuits. In this work, the capability of the RASP-FIT tool to generate a compilable faulty code is highlighted, however, we have used these benchmark circuits to validate the proposed test approach (which is not discussed here).

1) Compilable Faulty Design: Fig. 9 shows the original design and the compilable faulty design of the simple circuit from ISCAS'85 benchmark circuits for illustration purpose. There are some points to be noted here:

1) In the original design, the module name is `c17`, whereas, in faulty design, the module name is changed to `c17_1`, which shows the first copy of the faulty design.
2) The output ports of a faulty design are renamed with the extension of `(_f1)` for the first copy of the faulty design and for the second copy `(_f2)` and so on. This is done for the comparison purpose in fault injection experiments with the fault-free design.
3) The fault selection port (i.e. select port) is added to the faulty copy as an input port, which is used to choose a particular fault for injection and its activation.
4) The selected fault is activated by assign a logic '1' value. For that purpose, a `wire fis` is added to the design.
5) The fault variables `f0,f1,...,fn` are used to assign the 'fis' value in an `always` block, so these variables must be declared as `reg` variables.
6) DEMUX-based FISA unit is added to select and activate the fault. When no fault is activated, the circuit performs the same operation as of the original design.
7) This tool is capable of injecting faults in a full design or in a partial design. The user can specify any number of copies, and this tool evenly distributes the number of faults in each copy of the design.

2) Timing Analysis for Gate-Level Designs: The RASP-FIT tool takes appropriate time to generate faulty designs. We have performed the experiments on the various gate-level benchmark circuits from ISCAS'85 and ISCAS'89. The complexity of design in terms of logic gates are described in Table IV and Table V for the ISCAS'85 and ISCAS'89 benchmark circuits, respectively. Also, these tables show the total number of faults injected in the design. The time taken by the RASP-FIT tool is measured (in Seconds) using the Matlab commands `(tic,toc)`, and described in the last column of the tables.

TABLE IV. TIME ANALYSIS TO GENERATE FAULTY MODELS OF ISCAS'85 GATE-LEVEL DESIGNS

S.No.	Gate-level benchmark circuits	No. of logic gates	Total faults	Time (in Seconds)
1	c17	6	12	0.2075
2	c432	160	336	0.4187
3	c499	202	408	0.4578
4	c880	383	729	0.5811
5	c1355	546	1064	1.068
6	c1908	880	1498	1.8334
7	c2670	1269	2152	4.8450
8	c3540	1669	2939	6.4805
9	c5315	2307	4386	13.011
10	c6288	2416	4800	21.935
11	c7552	3513	6145	37.504

TABLE V. TIME ANALYSIS TO GENERATE FAULTY MODELS OF ISCAS'89 GATE-LEVEL DESIGNS

S.No.	Gate-level benchmark circuits	No. of logic gates/FFs	Total faults	Time (in Seconds)
1	s1494	647/6	1399	1.503
2	s5378	2779/179	4391	8.723
3	s9234	5597/211	8182	21.68
4	s13207	7951/638	11803	82.09
5	s15850	9772/534	14179	118.188
6	s35932	16065/1728	29997	965.05
7	s38417	22179/1636	33664	577.50
8	s38584	19253/1426	34182	1306.90

B. Data-flow Abstraction Level Code

The EPFL benchmark circuits consist of 23 combinational logic circuits, written in a data-flow code style. These circuits are specifically designed for logic optimization but we are using them for fault injection/simulation approaches.

```
// Original design
module c17 (N1,N2,N3,N6,N7,N22,N23);

input N1,N2,N3,N6,N7;
output N22,N23;
wire N10,N11,N16,N19;

nand NAND2_1 (N10, N1, N3);
nand NAND2_2 (N11, N3, N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22, N10, N16);
nand NAND2_6 (N23, N16, N19);

endmodule
```

```
// Compilable faulty design
module c17_1 (select,N1,N2,N3,N6,N7,N22_f1
    ,N23_f1);
input N1,N2,N3,N6,N7;
output N22_f1,N23_f1;
wire N10,N11,N16,N19;
input[1:0] select;
wire fis=1;
reg f0,f1,f2,f3;
always @ (select)
  begin
    if (select == 2'd0) begin
      f0=fis;f1=0;f2=0;f3=0; end
    else if (select == 2'd1) begin
      f0=0;f1=fis;f2=0;f3=0; end
    else if (select == 2'd2) begin
      f0=0;f1=0;f2=fis;f3=0; end
    else if (select == 2'd3) begin
      f0=0;f1=0;f2=0;f3=fis; end
    else begin
      f0=0;f1=0;f2=0;f3=0; end
  end
nand NAND2_1 (N10,f0^ N1,f1^ N3);
nand NAND2_2 (N11,f2^ N3,f3^ N6);
nand NAND2_3 (N16, N2, N11);
nand NAND2_4 (N19, N11, N7);
nand NAND2_5 (N22_f1, N10, N16);
nand NAND2_6 (N23_f1, N16, N19);
endmodule
```

Fig. 9. Original code (left) & instrumented compilable design code (right) by RASP-FIT.

1) Compilable Faulty Design: A simple code of 4_to_1 multiplexer, written in a data-flow abstraction style is considered to present the output of RASP-FIT tool for data-flow designs. Fig. 10 shows the original design and instrumented faulty design of it. The detail description of the faulty code is described already in gate-level design. The difference lies in the way of the injection of faults in the design. Fig. 6 shows the method of injection in a single bit and vector variables for all abstraction levels.

2) Timing Analysis for Data-flow Designs: We performed the experiments on the various data-flow benchmark circuits from EPFL. The complexity of design in terms of logic gates, the total number of faults in the design, and time taken by the tool in Seconds are described in Table VI and Table VII.

C. Behavioural Abstraction Level Code

At this level, the large and complex designs are written e.g. processor design. In order to perform fault injection testing for these bigger design, a fault-free module is replaced by a generated faulty module. Different basic and large complex behavioural designs are considered from DP32 Verilog processor and presented in Table VIII. Instead of the number of logic gates information, we added the number of slices LUTs (Look-Up Tables) obtained after the synthesis process using Xilinx ISE tools. Fig. 11 shows the original design and instrumented faulty design of it. The way of fault injection mechanism (code parsing) for the different behavioural commands, e.g. case, if-else construct, blocking and non-blocking assignments, always-initial blocks is added in RASP-FIT. The fault injection mechanism for other behavioural commands such as loops, built-in or user-defined macros & functions, and the include

```
// Original design
module mux_4x1 (out, in0, in1, in2, in3, s0
    , s1);

output out;
input in0, in1, in2, in3;
input s0, s1;

assign out = (~s0 & ~s1 & in0)|
(s0 & ~s1 & in1) |
(~s0 & s1 & in2) |
(s0 & s1 & in0);
endmodule
```

```
// Compilable faulty design
module mux_4x1_1 (select,out_f1, in0, in1,
    in2, in3, s0, s1);
output out_f1;
input in0, in1, in2, in3;
input s0, s1;
input [3:0] select;
wire fis=1;
reg f0,f1,f2,f3;
always @ (select)
  begin
    if (select == 2'd0) begin
      f0=fis;f1=0;f2=0;f3=0; end
    else if (select == 2'd1) begin
      f0=0;f1=fis;f2=0;f3=0; end
    else if (select == 2'd2) begin
      f0=0;f1=0;f2=fis;f3=0; end
    else if (select == 2'd3) begin
      f0=0;f1=0;f2=0;f3=fis; end
    else begin
      f0=0;f1=0;f2=0;f3=0; end
    end
  assign out_f1 = ((f0 ^ ~s0) & (f1 ^ ~s1) &
      (f2 ^ in0)) |
  ((f3 ^ s0) & ~s1 & in1) |
  (~s0 & s1 & in2)|(s0 & s1 & in0);
endmodule
```

Fig. 10. Original code (left) & instrumented compilable data-flow code (right) by RASP-FIT.

TABLE VI. TIME TO GENERATE FAULTY MODULES OF ARITHMETIC DATA-FLOW CIRCUITS FROM EPFL

S.No.	Data-flow benchmark circuits	No. of logic gates	Total faults	Time (in Seconds)
1	Adder	1020	2040	6.512
2	Barrel-shifter	3336	6672	89.898
3	Divisor	44762	114494	4034.22
4	Hypotenuse	214335	428670	-
5	Log2	32060	64120	794.34
6	Max	2865	5730	41.26
7	Multiplier	27062	54124	1153.9
8	Sine	5412	10832	62.93
9	Square	24618	36970	955.66
10	Square-root	18484	49236	1212.9

TABLE VII. TIME TO GENERATE FAULTY MODULES OF RANDOM/CONTROL DATA-FLOW CIRCUITS FROM EPFL

S.No.	Data-flow benchmark circuits	No. of logic gates	Total faults	Time (in Seconds)
1	Round-Robbin arbieter	11839	23678	311.659
2	ALU control unit	174	349	0.484
3	Coding-cavlc	693	1386	2.177
4	Decoder	304	608	1.393
5	I2C controller	1342	2699	10.61
6	Int-to-float controller	260	520	1.601
7	Memory controller	46836	93946	15794
8	Priority encoder	978	1956	3.396
9	look-ahead xy router	257	541	0.806
10	voter	13758	27516	152.30

files are in progress.

```
// Original design
module mux_4x1 (out, in0, in1, in2, in3,
    sel);
output reg out;
input in0, in1, in2, in3;
input [1:0] sel;

always @ (in0 or in1 or in2 or in3 or sel)
  begin
    case (sel)
    2'b00: out = in0;
    2'b01: out = in1;
    2'b10: out = in2;
    2'b11: out = in3;
    endcase
  end
endmodule

// Compilable faulty design
module mux_4x1 (select,out, in0, in1, in2,
   in3, sel);
output reg out_f1;
input in0, in1, in2, in3;
input [1:0] sel;
input [2:0] select;
wire fis=1;
reg f0,f1,f2,f3,f4,f5;
always @ (select)
  begin
    if (select == 3'd0) begin
      f0=fis;f1=0;f2=0;f3=0;f4=0;f5=0; end
    else if (select == 3'd1) begin
      f0=0;f1=fis;f2=0;f3=0;f4=0;f5=0; end
    .
    .
    .
    else begin
      f0=0;f1=0;f2=0;f3=0;f4=0;f5=0; end
  end

always @ (in0 or in1 or in2 or in3 or sel)
  begin
    case ({f0,f1} ^ sel)
      2'b00: out = f2 ^ in0;
      2'b01: out = f3 ^ in1;
      2'b10: out = f4 ^ in2;
      2'b11: out = f5 ^ in3;
    endcase
  end
endmodule
```

Fig. 11. Original code (left) & instrumented compilable behavioural code (right) by RASP-FIT.

VI. Conclusion

Fault S/E helps designers and test engineers in the evaluation, verification of their designs and generation of test patterns. It is used to evaluate fault effects, dependability and measure the robustness of FPGA-based systems, written in HDL. The injection of faults in HDL design requires modification of design to generate faulty target system. In this work, the code modifier for Verilog HDL designs is presented in detail. The RASP-FIT is a fault injection tool, which works at the code level of the designs at various abstraction levels. This tool can inject faults in the whole design, and produce the compilable code. The tool is simple, automatic and user-friendly. Results show that the RASP-FIT tool takes an appropriate time, depends on the size of the design, for the generation of faulty module and fault injection control unit.

References

[1] Z. Navabi, *Digital System Test and Testable Design*. Boston, MA: Springer US, 2011.

[2] H. Ben Fekih, A. Elhossini, and B. Juurlink, *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Cham: Springer International Publishing, 2015, vol. 9040.

TABLE VIII. TIME TO GENERATE FAULTY MODULES OF BEHAVIOURAL DESIGNS

S.No.	Behavioural designs	No. of slices LUTs	Total faults	Time (in Seconds)
1	Mux (case)	1	6	0.108
2	Mux (if-else)	1	12	0.112
3	8-bit Full Adder	9	75	0.648
4	Program Counter	12	130	0.785
5	ALU-32bit	173	896	3.359

[3] A. R. Khatri, A. Hayek, and J. Börcsök, *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, V. Bonato, C. Bouganis, and M. Gorgon, Eds. Cham: Springer International Publishing, 2016, vol. 9625.

[4] A. R. Khatri, A. Hayek, and J. Börcsök,"Validation of selecting SP-values for fault models under proposed RASP-FIT tool," in *2017 First International Conference on Latest trends in Electrical Engineering and Computing Technologies (INTELLECT)*. Karachi, Pakistan: IEEE, pp. 1–7, Nov 2017.

[5] A. R. Khatri, A. Hayek, and J. Börcsök, "Validation of the Proposed Fault Injection , Test and Hardness Analysis for Combinational Dataflow Verilog HDL Designs under the RASP-FIT Tool," in *2018 IEEE 16th Int. Conf. on Dependable, Autonomic & Secure Comp., 16th Int. Conf. on Pervasive Intelligence & Comp., 4th Int. Conf. on Big Data Intelligence & Comp., and 3rd Cyber Sci. & Tech. Cong.* Athens, Greece: IEEE Comput. Soc, pp. 544–551, Aug 2018.

[6] L. Entrena, "Fast fault injection techniques using FPGAs," in *14th Latin American Test Workshop - LATW*. Madrid,Spain: IEEE, Apr 2013.

[7] A. Benso, M. Rebaudengo, M. Reorda, and P. Civera, "An integrated HW and SW fault injection environment for real-time systems," in *Proceedings IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (Cat. No.98EX223)*. Austin, TX, USA, USA: IEEE Comput. Soc, pp. 117–122, 1998.

[8] D. Kammler, J. Guan, G. Ascheid, R. Leupers, and H. Meyr, "A Fast and Flexible Platform for Fault Injection and Evaluation in Verilog-Based Simulations," in *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*. IEEE, pp. 309–314, Jul 2009,

[9] A. Benso and P. Prinetto, *Fault Injection Techniques And Tools For Embedded Systems Reliability Evaluation.* Kluwer Academic Publishers, 2003.

[10] M. Kooli and G. Di. Natale, "A survey on simulation-based fault injection tools for complex systems", in *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. Santorini: IEEE, pp. 1–6, May 2014.

[11] V. Sieh, O. Tschache, and F. Balbach, "VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions," in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. IEEE Comput. Soc,pp. 32–36, 1997.

[12] H. Ziade, R. Ayoubi, and R. Velazco, "A Survey on Fault Injection Techniques," *The International Arab Journal of Information Technology*, vol. 1, no. 2, pp. 171–186, 2004.

[13] W. Chao, F. Zhongchuan, C. Hongsong, and C. Gang, "FSFI: A Full System Simulator-Based Fault Injection Tool," in *2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control*. IEEE, pp. 326–329, Oct 2011.

[14] W. Mansour, R. Velazco, R. Ayoubi, H. Ziade, and W. El Falou, "A method and an automated tool to perform SET fault-injection on HDL-based designs," in *25th International Conference on Microelectronics (ICM)*. Beirut: IEEE, pp. 1–4, Dec 2013.

[15] W. Mansour, M. A. Aguirre, H. Guzman-Miranda, J. Barrientos, and R. Velazco, "Two complementary approaches for studying the effects of SEUs on HDL-based designs," in *IEEE 20th International On-Line Testing Symposium (IOLTS)*. IEEE, pp. 220–221, Jul 2014.

[16] W. Mansour and R. Velazco, "An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2728–2733, Aug 2013.

[17] W. Mansour and R. Velazco, "SEU Fault-Injection in VHDL-Based Processors: A Case Study"*Journal of Electronic Testing*, vol. 29, no. 1, pp. 87–94, Feb 2013.

[18] M. Shokrolah-Shirazi and S. G. Miremadi, "FPGA-Based Fault Injection into Synthesizable Verilog HDL Models," in *Second International Conference on Secure System Integration and Reliability Improvement*. Yokohama: IEEE, pp. 143–149, Jul 2008.

[19] B. Rahbaran, A. Steininger, and T. Handl, "Built-in Fault Injection in Hardware - The FIDYCO Example," in *Second IEEE International Workshop on Electronic Design, Test and Applications*. Perth, WA, Australia: IEEE, pp. 327–327, 2004.

[20] M. Jeitler, M. Delvai, and S. Reichor, "FuSE - a hardware accelerated HDL fault injection tool," in *2009 5th Southern Conference on Programmable Logic (SPL)*. Sao Carlos: IEEE, pp. 89–94, Apr 2009.

[21] A. Mohammadi, M. Ebrahimi, A. Ejlali, and S. G. Miremadi, "SCFIT: A FPGA-based fault injection technique for SEU fault model," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Dresden: IEEE, pp. 586–589, Mar 2012.

[22] L. Naviner, J.-F. Naviner, G. dos Santos, E. Marques, and N. Paiva, "FIFA: A fault-injection–fault–analysis-based tool for reliability assessment at RTL level," *Microelectronics Reliability*, vol. 51, no. 9-11, pp. 1459–1463, Sep 2011.

[23] A. Rohani and H. G. Kerkhoff, "Rapid transient fault insertion in large digital systems," *Microprocessors and Microsystems*, vol. 37, no. 2, pp. 147–154, Mar 2013.

[24] C. Dunbar and K. Nepal, "Using Platform FPGAs for Fault Emulation and Test-set Generation to Detect Stuck-at Faults," *Journal of Computers*, vol. 6, no. 11, pp. 2335–2344, Nov 2011.

[25] C. Pohl, C. Paiz, and M. Porrmann, "vMAGIC–Automatic Code Generation for VHDL," *International Journal of Reconfigurable Computing*, vol. 2009, pp. 1–9, 2009.

[26] Joseph Cavanagh, *Digital Design Verilog and HDL Fundamentals*. Taylor and Francis Group, LLC, 2011.

[27] S. Palnitkar, *Verilog HDL A guide to Digital Design and Synthesis*. SunSoft Press, 1996.

[28] Sponsored by the Design Automation Standards Committee, *IEEE Standard for Verilog Hardware Description Language*, vol. 2005, no. April 2006.

[29] A. R. Khatri, M. Milde, A. Hayek, and J. Börcsök, "Instrumentation Technique for FPGA based Fault Injection Tool," in *5th International Conference on Design and Product Development (ICDPD'14)*, Istanbul, Turkey, pp. 68–74, Dec 2014.

[30] R. Drechsler, S. Eggersglüß, G. Fey, and D. Tille, *Test Pattern Generation using Boolean Proof Engines*. Springer Science+Business Media, 2009.