Noname manuscript No.
(will be inserted by the editor)

Simulating the effects of logic faults in implementation-level VITAL-compliant models

Ilya Tuzov · David de Andrés · Juan-Carlos Ruiz

Received: date / Accepted: date

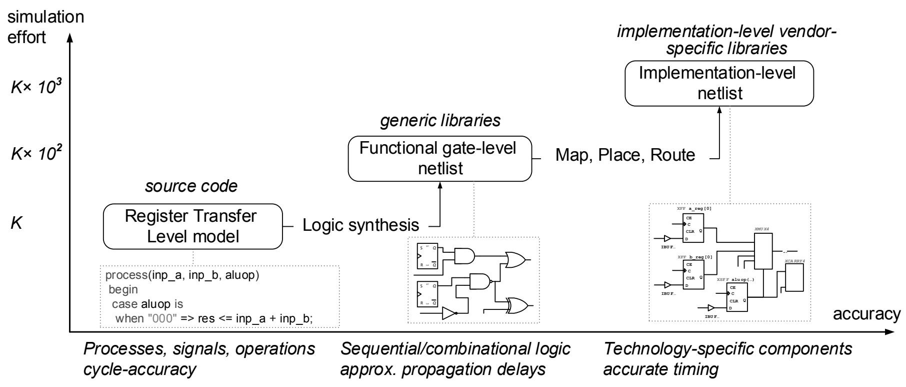
Abstract Simulation-based fault injection (SBFI) is a well-known technique to assess the dependability of hardware designs specified using Hardware Description Languages (HDL). Although logic faults are usually introduced in models defined at the Register Transfer Level (RTL), most accurate results can be obtained by considering implementation-level ones, which reflect the actual structure and timing of the circuit. These models consist of a list of interconnected technology-specific components (macrocells), provided by vendors and annotated with post-place-and-route delays. Macrocells described in the Very High Speed Integrated Circuit HDL (VHDL) should also comply with the VHDL Initiative Towards Application Specific Integrated Circuit Libraries (VITAL) standard to be interoperable across standard simulators. However, the rigid architecture imposed by VITAL makes that fault injection procedures applied at RTL cannot be used straightforwardly. This work identifies a set of generic operations on VITAL-compliant macrocells that are later used to define how to accurately simulate the effects of common logic fault models. The

This work has been partially funded by the Ministerio de Economía, Industria y Competitividad of Spain under grant agreement no TIN2016-81075-R, and the "Programa de Ayudas de Investigación y Desarrollo" (PAID) of Universitat Politècnica de València.

Ilya Tuzov
ITACA, Universitat Politècnica de València, Campus de Vera s/n, 46022, Spain
Tel.: +34 963877007
E-mail: tuil@disca.upv.es
ORCID 0000-0002-1980-0708

David de Andrés
ITACA, Universitat Politècnica de València, Campus de Vera s/n, 46022, Spain
E-mail: ddandres@disca.upv.es
ORCID 0000-0002-4744-3795

Juan-Carlos Ruiz
ITACA, Universitat Politècnica de València, Campus de Vera s/n, 46022, Spain
E-mail: jcruizg@disca.upv.es
ORCID 0000-0001-7678-3513


simulation effort
K× 10^3
K× 10^2
K
implementation-level vendor-specific libraries
Implementation-level netlist
generic libraries
Functional gate-level netlist
Map, Place, Route
source code
Register Transfer Level model
Logic synthesis
accuracy
Processes, signals, operations cycle-accuracy
Sequential/combinational logic approx. propagation delays
Technology-specific components accurate timing

```
process(inp_a, inp_b, aluop)
 begin
  case aluop is
   when "000" => res <= inp_a + inp_b;
```

Fig. 1: Simulation effort at different HDL description levels

generality of this proposal is supported by the definition of a platform-specific fault procedure based on these operations. Three embedded processors, implemented using the Xilinx's toolchain and SIMPRIM library of macrocells, are considered as a case study, which exposes the gap existing between the robustness assessment at both RTL and implementation-level.

Keywords Simulation-based fault injection · implementation-level HDL models · VITAL · semicustom design flow

1 Introduction

Modern semicustom hardware design flow relies on the use of Hardware Description Languages (HDL) to model hardware at either the implementation level, the logic level, or the Register Transfer Level (RTL). Nevertheless, hardware models are usually defined at the much simpler and higher-level RTL, which describes the circuit in terms of registers and how information flows among them with just clock cycle accuracy. It is the task of Electronic Design Automation (EDA) tools to refine the RTL model into a gate level netlist representation and, after that, an implementation level model for the selected implementation technology [34], as depicted in Fig. 1. Accordingly, accurate performance, area, power consumption, and dependability estimations, can only be obtained from detailed implementation-level models [30].

EDA tools also enable the simulation of these models to verify both their functionality and timing behavior [27]. This verification process, which is mandatory previous to manufacturing any circuit, must also be accompanied by a dependability assessment process whenever unexpected failures in the system may lead to human lives or economic losses, environmental damage, or affect the manufacturer's reputation. Simulation-based fault injection (SBFI) [4] is a privileged technique to intentionally introduce faults at simulation run-time for the dependability assessment and verification of systems described using HDL models. This technique is usually applied to RTL models due to the uniformity of the involved procedures, and the low computational cost of the simulation process. As depicted in Fig. 1, the simulation complexity growth drastically as the model approaches to implementation level, being 2 to 4 orders of magnitude slower than at RTL [29]. On the one hand, simulation components provided by device vendors (known as macrocells) describe very accurately the timing behaviour of the components. An inverter which is defined as 1 line in RTL ($a <= not\ b$) is translated into the macrocell described in Listing 1 with 40 lines of code (removing comments and blank lines), without considering the code of required libraries. On the other hand, a simple combinational component like an 8-bit adder which is defined in 1 line in RTL ($result <= a + b$) is translated into a set of 34 interconnected macrocells (16 input buffers, 8 output buffers, 8 look-up tables and 2 carry chains).

SBFI approaches rely on the modification/instrumentation of the HDL model and/or on simulator commands [9].

Those that instrument the model use *saboteurs*—components that alter the behavior/timing of signals—, and *mutants*—variations of existing components that reproduce their behaviour in presence of faults. Most notable instrumentation-based tools include: i) VERIFY [28], which makes the provider responsible for taking into account all possible faults in the provided macrocells and requires a non-standard HDL simulator to support an extension of the VHDL language; ii) the proposal in [16], which instruments the original library for post-synthesis simulation to enable the injection of SEUs in FPGA-based designs; and iii) a saboteur-based technique [3], which takes into account the setup/hold time window to properly simulate bit-flips in clock-gated ASICs. Although these methods enable the injection of sophisticated fault models [1], they are highly intrusive, as they introduce/replace components into/from the original model. Moreover, they cause a significant overhead in the experimental process. If the model is instrumented for each experiment, which may take a couple of minutes for a medium-sized netlist (100K gates), this means that the experimentation will take 2 additional weeks to complete for a campaign consisting of 10000 experiments. If the model is instrumented to support all the faults that have to be injected in the campaign, then the size of the model will increase so much that will slow down the simulation of each experiments.

Simulator commands avoid instrumenting the model by changing the value of signals/variables at run-time to simulate the effect of faults [4]. Most notable tools supporting the SBFI by simulator commands are MEFISTO [14] and VFIT [2] tools. An alternative command-based approach [7] injects stuck-at faults into Verilog models by including Verilog instructions in the testbench. The approach proposed in [20] injects logic faults at gate-level with a rate dependent on the switching activity of the netlist. Even if the set of logic fault models that can be injected using simulator commands is smaller than instrumenting the model, this is the preferred technique to reduce the overhead in the simulation time. However, the same approaches used for RTL models cannot be applied to implementation-level ones, as simulator commands sequences do not take into account the particular architecture of macrocells, which results in an unexpected behaviour of the target component in presence of faults [33].

Alternative SBFI approaches proposed the development of non-standard simulators to speed-up the simulation of faults, like the one targeting IcarusVerilog models [19] or those using GPGPUs [15]. However, these approaches are not generic and cannot be applied to off-the-shelf EDA tools.

This contribution aims at defining a generic approach, which could be applied to any standard simulator and implementation target, for enabling the simulation of logic faults at implementation-level models described using the Very High Speed Integrated Circuit HDL (VHDL) [13]. These macrocells must comply with the VHDL Initiative Towards ASIC Libraries (VITAL) standard [12] to make them portable across EDA tools, enable the back-annotation of timing properties, and enable simulators to implement specific optimizations. Although comprehensive guidelines for developing VITAL-based components exist [18] [6], works focused on the injection of faults in these macrocells did not take into account the accuracy of the considered fault models [8], and generated highly intrusive injectable VITAL-compliant models for a single type of logic faults [25] [26]. Authors previous work on this subject [33] focused on how to inject stuck-at and delay faults into Xilinx's SIMPRIM macrocells from a very practical perspective. This contribution extends this work by analysing the architecture of VITAL-compliant macrocells and determining which generic operations would be required to support the injection of logic faults into these components. These operations are defined so that simulator commands are employed whenever possible, reducing the instrumentation of the macrocell to a minimum. The proposed operations keep the functional and timing behaviour of the target component, have a negligible impact on the simulation time, and ensure that VITAL requirements are met. Only the injection of interconnect path delays will degrade the model from VITAL level 1 to 0. By targeting the vendor's macrocells neither the original implementation-level model nor the VITAL libraries are modified and no further recompilation will be required in future experiments.

The rest of the paper is structured as follows. Section 2 describes in depth the architecture of VITAL-compliant macrocells. Section 3 analyses this architecture to define a number of generic operations to support the injection of faults VITAL-compliant macrocells. High-level procedures based on these operations are derived for most common logic fault models. The generality of this proposal is shown by defining a platform-specific fault injection procedure. The case study described in Section 4 details an experimental procedure that follows the defined generic operations. Results analysed in Section 5 illustrate the gap existing between results obtained at RTL and implementation level, and the importance of defining procedures for accurate robustness assessment. Finally, Section 6 presents the main conclusions of this work.

2 The VITAL Standard

In its early years, there was no uniform and efficient method for handling timing in VHDL, which resulted in a lack of ASIC libraries with which model

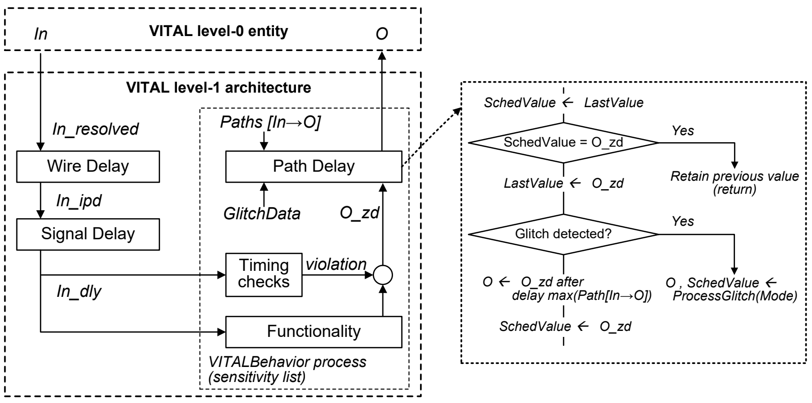


Fig. 2: VITAL-compliant macro-cell model

and implement digital systems. The VHDL Initiative Towards ASIC Libraries (VITAL) standard [12] was the result of an agreement among ASIC vendors, EDA tool vendors, and ASIC designers about the requirements (timing accuracy, model maintainability, and simulation performance) for the effective modelling of ASIC primitive, or macrocells, in VHDL.

The VITAL specification contains four main elements: i) the *Model Development Specification document* defines how to specify ASIC libraries in VITAL-compliant VHDL to be used in simulators; ii) the *Vital_Timing package* provides a standard set of procedures for checking timing constraints defined in a Standard Delay Format (SDF) [24] file; iii) the *Vital_Primitives package* models all gate-level primitives already used by simulation tools vendors, so they could be optimised for the faster simulation of VHDL; and iv) the *VITAL SDF map* which maps SDF files to VHDL generic values.

The basic architecture of a VITAL-compliant macrocell is depicted in Fig. 2 and Listing 1 specifies an inverter that follows this architecture.

VITAL defines two levels of support. *VITAL level 0* requires the definition of a level 0 attribute (line 21 of Listing 1), using ports of type *std_ulogic* and *std_logic_vector* with no underscores in the port names (lines 18–19), and special naming convention for timing generics (lines 8–11, with ports names prefixed as *tpid_*—interconnect path delay that represents the delay between components—and *tpd_*—propagation delay that represents the pin-to-pin delay within a component). Compliance with VITAL level 0 provides SDF back annotation and negative timing constraints. *VITAL level 1* requires the definition of a level 1 attribute (line 26), no use of shared variables, use of those operators defined in the *Standard* and *std_logic_1164* packages (lines 1–3 ensure that only those packages, in addition to the VITAL packages, are used), and all outputs must be driven by a *VitalPathDelay* or a *Vital* primitive (lines 53–63 make use of a *VitalPathDelay01* primitive to drive the *YNeg* output—line 54). Compliance with VITAL level 1, as the inverter described in Listing 1, provides accelerated simulation of primitives and tables.

Any VITAL-compliant macrocell must include a *Wire Delay* block, a *Signal Delay* block, and a *VITALBehavior process*, as depicted in Fig. 2.

Interconnect delays represent the time it takes a signal to traverse the circuit from one component to another, and it will depend on various factors, like the length of the wire, its resistance, and fan-out, among others. As there is no way to simulate a wire in VHDL, the *Wire Delay* block (lines 30–34 in Listing 1) is in charge of delaying incoming signals by the time specified in the associated timing generic (*tpid* using the *VitalWireDelay* routine (line 37 in Listing 1). This routine can only be called once per input port, which cannot be used anywhere else in the model afterwards, and its output must be an internal signal (*A_ipd* in the example).

Components may present negative timing constraints (either setup or hold times, not both) only if they present some kind of internal delay. The *Signal Delay* block takes charge of delaying the internal signals of the component (suffix *_ipd*), if needed, in a similar fashion to the *Wire Delay* block, but using the *VitalSignalDelay* routing instead. The sample model has no negative timing constraints, so this block is not defined (line 36).

The actual functionality of the component is defined within a process labelled as *VITALBehaviour* (lines 38–63 in Listing 1). All signal read within the process (*A_ipd* is read in line 46) must be included in its sensitivity list (line 39), so the process will be triggered whenever the value of any of these signals changes. First of all, the process may perform timing constraint checks for possible violations if the *TimingChecksOn* generic parameter is active. Those checks make use of predefine routines, like *VitalSetupHoldCheck*, from the *VITAL_Timings* package. No timing checks are performed in the sample model (line 47). After that, the functionality section computes the actual logic function of the component without any delay (*YNeg_zd*). Lines 49-50 of Listing 1 define the behaviour of the inverter by means of the predefined *VitalINV* function to comply with VITAL level 1. Finally, the output values computed are delayed after applying the appropriate delays using *VitalPathDelay* procedures (lines 52-63). These procedures enable different optimisations of the simulation like, for instance, checking whether the output has changed with respect to the previously scheduled value to prevent further processing.

3 Injecting Logic Faults into VITAL-compliant Macrocells

Commonly used fault injection approaches at RTL cannot be directly applied to implementation level VITAL-compliant models. This section defines generic operations on those macrocells to support the injection of logic faults and how to use them to mimic the expected effect of the selected fault models.

Listing 1: VITAL-compliant inverter gate (std04.vhd) [22]

```vhdl
LIBRARY IEEE; USE IEEE.std_logic_1164.ALL;
              USE IEEE.VITAL_timing.ALL;
              USE IEEE.VITAL_primitives.ALL;

-- ENTITY DECLARATION
ENTITY std04 IS
  GENERIC (
    -- tipd delays: interconnect path delays
    tipd_A       : VitalDelayType01 := VitalZeroDelay01;
    -- tpd delays
    tpd_A_YNeg   : VitalDelayType01 := UnitDelay01;
    -- generic control parameters
    MsgOn        : BOOLEAN          := DefaultMsgOn;
    XOn          : BOOLEAN          := DefaultXOn;
    InstancePath : STRING           := DefaultInstancePath
  );
  PORT (
    A            : IN  std_ulogic   := 'U';
    YNeg         : OUT std_ulogic   := 'U'
  );
  ATTRIBUTE VITAL_LEVEL0 of std04 : ENTITY IS TRUE;
END std04;

-- ARCHITECTURE DECLARATION
ARCHITECTURE vhdl_behavioral of std04 IS
  ATTRIBUTE VITAL_LEVEL1 of vhdl_behavioral : ARCHITECTURE IS TRUE;
  SIGNAL    A_ipd : std_ulogic := 'U';

BEGIN
  -- Wire Delays
  WireDelay : BLOCK
  BEGIN
    w_1: VitalWireDelay (A_ipd, A, tipd_A);
  END BLOCK;

  -- No Signal Delay block

  -- VITALBehavior Process
  VITALBehavior : PROCESS(A_ipd)

    -- Functionality Results Variables
    VARIABLE YNeg_zd      : std_ulogic          := 'U';
    -- Output Glitch Detection Variables
    VARIABLE Y_GlitchData : VitalGlitchDataType;

  BEGIN
    -- No Timing Checks section

    -- Functionality Section
    YNeg_zd := VitalINV (A_ipd);

    -- Path Delay Section
    VitalPathDelay01 (
      OutSignal     => YNeg,
      OutSignalName => "YNeg",
      OutTemp       => YNeg_zd,
      XOn           => XOn,
      MsgOn         => MsgOn,
      Paths         => (
        0 => (InputChangeTime => A_ipd'LAST_EVENT,
              PathDelay       => tpd_A_Yneg,
              PathCondition   => TRUE)),
      GlitchData    => Y_GlitchData );

  END PROCESS;

END vhdl_behavioral;
```

Table 1: Operations on VITAL-compliant macrocells to support fault injection

Operation	Type	Description
examine(target)	simulator command	gets the current value of the *target* signal or variable
force(target, mode, value, duration)	simulator command	changes the state of the *target* signal to *value* for *duration* (*freeze mode*) or until overwritten (*deposit mode*)
change(target, value)	simulator command	like the *force* command, but the *target* is a constant, generic, or variable
generic2signal(target)	instrumentation rule	initialises an internal signal with the value of the generic and feeds that signal wherever the generic is used
constant2signal(target)	instrumentation rule	like the *generic2signal* operation, but the target is a constant
addToList(target, signal)	instrumentation rule	adds the *signal* signal to the sensitivity list of the process identified by the *target* label
encloseInProcess(target)	instrumentation rule	encloses the procedure identified by the *target* label into a process activated by all the incoming parameters of the enclosed procedure

3.1 Definition of Generic Operations to Support the Fault Injection

The netlist obtained after the synthesis of an RTL model described using VHDL consists of a set of interconnected VITAL-compliant macrocells. As these macrocells are also defined in VHDL it is possible to follow the *simulator commands* approach [4] to modify the state of its internal signals and variables to reproduce the effect of a given fault model and observe the behaviour of the system in presence of such fault. Table 1 lists the commonly used operations by state of the art simulators to retrieve and modify the contents of signals and variables of the model. These commands have been defined after Mentor Graphics' Modelsim/Questasim commands [17]. So, taking Listing 1 as a reference, the current state of signal *A_ipd* can be obtained by the *examine(A_ipd)* operation, and the state of the *YNeg_zd* variable can be set to a low logic level by the *change(YNeg_zd, 0)* operation. Nevertheless, not all possible elements of a macrocell can be directly modified by following this approach, thus requiring the definition of additional generic operations to support the required transformations in the VITAL-compliant model.

Sometimes would be interesting to modify the value of generic parameters of a model, like the specified delays, the logic function implemented by a look-up table, or the initial contents of a memory block. However, as noted in Questsim commands reference manual [17], changes in generic parameters cannot take place if the design is optimised for fast simulation and, even so, the simulation will not recompute dependent expressions. Thus, a new operation called *generic2signal* has been defined to include supplementary signals in the macrocell that can capture the value of generic parameters and feed that signals to those elements using the associated generics. Thus, to make injectable the *tipd_A* generic from Listing 1, the *generic2signal(tpid_A)* operation should be executed. It will take charge of: i) creating a new signal of the same type as the generic in the declarative part of the model architecture (lines 26–28)—*SIGNAL v_tpd_A_YNeg : VitalDelayType01 := UnitDelay01;* ii) initialising that signal after back-annotation by including the following assignment—*v_tpd_A_YNeg <= tpd_A_YNeg*—outside any other block defined in the body of the model (lines 30–66); and iii) using that signal instead of the generic wherever required, like in line 61—*PathDelay =>v_tpd_A_YNeg.*

A similar procedure but handling constants is defined by *constant2signal.*

Listing 2: Enabling procedures to recompute incoming generics and constants

```
  encloseInProcess_w_1 : PROCESS(A, v_tipd_A)
  BEGIN
    w_1: VitalWireDelay (A_ipd, A, v_tipd_A);
  END PROCESS;
```

Processes are only activated upon changes on any of the signals listed in their sensitivity list. Thus, the transformation of any generic or constant into an internal signal that is used within a process requires also this signal to be included into the process sensitivity list. The *addToList* operation takes care of this transformation. Following the previous example, the *addToList(VITALBehavior, v_tpd_A_YNeg)* operation will include the signal *v_tpd_A_YNeg* in the sensitivity list of the *VITALBehavior* process in line 39—*VITALBehavior : PROCESS(A_ipd, v_tpd_A_YNeg)*.

VITAL level 1 enables the deployment of optimisations to speed up the simulation of the model. For instance, when procedures have input parameters taken from generics and constants they are not rechecked during simulation, as they are not supposed to change dynamically. Thus, any fault injected in these elements will not propagate through the model. A new operation called *encloseInProcess* has been defined to insert the desired procedure within a process that will be activated whenever an input parameter changes its value. This will ensure that the new state of generics or constants (previously transformed by *generic2signal* or *constant2signal* operations) will be recomputed within the process. So, for the *VitalWireDelay* procedure call in line 33 to be aware of a change in its input *v_tipd_A* (resulting from the transformation of *tipd_A* generic into an internal signal), the operation *encloseInProcess(w_1)* will transform it into the code listed in Listing 2. It must be noted that, although this process describes exactly the same functionality as the bare procedure, it does not comply with the requirements for VITAL level 1, as a *Wire Delay* block can only contain concurrent procedure calls. The instrumented model still will be compliant with VITAL level 0 and could deploy most of the optimisations available for VITAL level 1, although this particular one, regarding the generics and constants, will be forfeited.

3.2 Injecting Logic Faults Models into VITAL-compliant Macrocells

Common logic faults [10] that can be injected at VHDL models using simulator commands include *stuck-at* (fixing the logic state of an element), *bit-flip* (inverting the logic state of a sequential element), *pulse* (inverting the logic state of a combinational element), *indetermination* (fixing the logic state of an element between the high and low logic levels), and *delay* (modifying the propagation delay of an element). Logic faults that target the interconnection

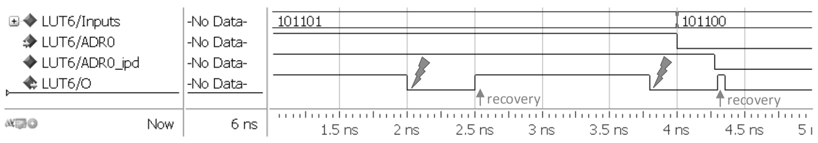


Fig. 3: Injecting two consecutive pulses into a combinational component

of the components, like *stuck-open*, *short*, *open-line*, and *bridging*, require the instrumentation of the model and are, thus, out of the scope of this study.

3.2.1 Stuck-at, Pulse, and Indetermination Faults

The permanent nature of stuck-at faults makes that their injection into implementation level models could follow exactly the same procedure used for RTL models. In this case, it is just a matter of targeting the signal connected to the output of the selected macrocell instead of dealing with the internal complexities of the component. As the signal will be permanently set to the injected value, it does not matter whether the internal state of the macrocell is really stuck or not. Thus, causing a stuck-at-1 to a flip-flop driving a signal named *ff_o* will be accomplished by using the operation *force(ff_o, freeze, 1)*.

As pulse fault models affect combinational components, and these macrocells do not store any logic value, it is also possible to just target the signal driven by the macrocell. For instance, a pulse can be injected for 10 ns to a look-up-table driving a signal named *lut_o* by means of these operations *newValue = (examine(lut_o) == 0) ? 1 : 0; force(lut_o, freeze, newValue, 10 ns)*. Fig. 3 displays the injection of two consecutive pulses in the output of an *X_LUT6* macrocell from the Xilinx's SIMPRIM library [35].

Permanent indetermination faults can be treated as stuck-at faults, and transient indetermination targeting combinational elements can follow the same approach as pulses, but setting the target signal to an 'X' value.

3.2.2 Bit-flip Faults

Bit-flip faults can be injected by inverting the logic value of RTL signals that represent sequential elements, like flip-flops. However, when applying that same procedure to the signal driven by the output of the sequential macrocell, the observed behaviour is not the one expected. As depicted in Fig. 4a, once the fault has been injected, the flip-flop's state is not restored on the following rising clock edge. On the next simulation event after the fault injection, the *VITALBehavior* process is activated (see Fig. 2). As this is a rising edge active flip-flop, the *Functionality* section recomputes the 'zero-delay' output. However, as *O_zd* equals the previously scheduled value (*Path Delay* checks), the output retains its current (faulty) value to optimize the simulation.

To bypass this check, the fault injection procedure should also invert the value of the *O_zd* variable. If no event occurs after the injection, the next rising

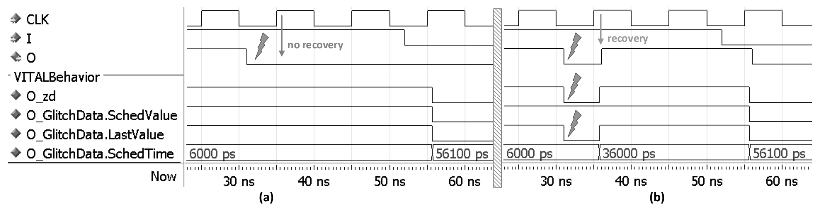


Fig. 4: Injecting a bit-flip fault on a flip-flop at implementation level following (a) the RTL procedure, and (b) the proposed approach

Listing 3: Proposed procedure to inject a bit-flip in the *target* macrocell

```
bitflip(target) {
   newValue = (examine(target/O) == 0) ? 1 : 0;
   force(target/O, deposit, newValue, 0);
   change(target/VITALBehavior/O_zd, newValue);
   change(target/VITALBehavior/O_GlitchData.LastValue, newValue);
}
```

clock edge updates the logic value on *O_zd* to '1'. As this is exactly the value that was previously scheduled, the output will retain its faulty logic value. Thus, the scheduled value should also be targeted to prevent this erroneous behaviour. However, the first action within the *Path Delay* section updates this value according to the last value stored to prevent glitches. Hence, the *O_GitchData.LastValue* variable should have also been modified.

Now, as depicted in Fig. 4b, the flip-flop recovers from the fault as expected. On the rising clock edge, the scheduled value is updated with the last value and, as being different from that captured on the clock edge, *O_zd* is propagated to the scheduled and last values and the output of the flip-flop. The proposed sequence of simulator commands-based operations is listed in Listing 3.

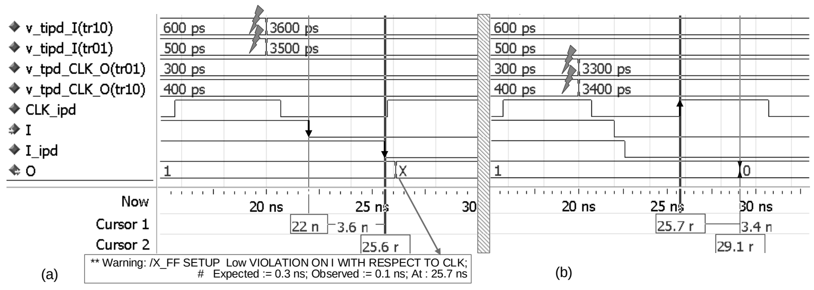
3.2.3 Delay Faults

Although possible in theory, delays faults are rarely considered in RTL models, whereas implementation level models present a very accurate timing description that enables the injection of these faults.

All VITAL-compliant macrocells must define two main timing generic parameters: interconnect path delays and propagation delays. Being generic parameters, it will be necessary to include some internal signals to be able to modify their values and pass these signals to the elements that use the original generics. In the case of generics used in the *Path Delay* block, this signal must be added to the sensitivity list of the *VITALBehavior* process. If the generics are used in the *Wire Delay* block, then it must be enclosed into a process activated by all its incoming signals. The proposed procedure for instrumenting a macrocell to support both types of delays is listed in Listing 4.

Listing 4: Enabling the injection of delays in the *target* macrocell

```
delayInstrumentation(target) {
   foreach(genericInput in target) {
      if (prefix(genericInput, "tpd_") {
         internalSignal = generic2signal(genericInput);
         addToList("VITALBehaviour", internalSignal);
      }
      else if (prefix(genericInput, "tipd_") {
         blockLabel = generic2signal(genericInput);
         encloseInProcess(blockLabel);
      }
   }
}
```



** Warning: /X_FF SETUP Low VIOLATION ON I WITH RESPECT TO CLK;
Expected := 0.3 ns; Observed := 0.1 ns; At : 25.7 ns

Fig. 5: Injecting delay faults into a flip-flop: a) interconnect delay of I input, and b) propagation delay from CLK to O path

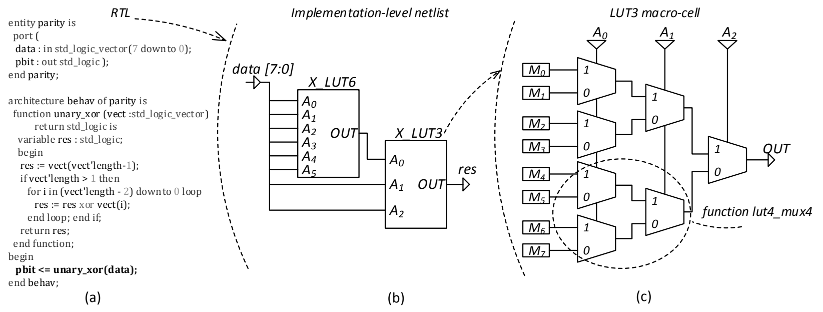
Once the macrocell instrumented, the available operations based on simulator commands can be used to change the state of those internal signals holding the delay values. For instance, after instrumenting the inverter modelled in Listing 1 by means of *delayInstrumentation(std04)*, the propagation delay can be increased in 2 ns by calling *force(v_tpd_A_YNeg, freeze, 2 ns, 0)*.

Fig. 5a depicts a warning issued by the simulator due to a setup time violation after increasing the interconnect delay of the *I* input port of a flip-flop from the Xilinx's SIMPRIM library [35] in 3 ns. Likewise, Fig. 5b shows the case of increasing the propagation delay from *CLK* to *O* in 3 ns.

3.3 Generality of this Proposal: Considering Device-specific Components

The proposed procedures are generic enough to be applied to any VITAL-compliant macrocell and support the injection of any logic fault not related to the interconnection of components. To show its generality, these operations will be used to define a platform-specific fault injection approach to enable the injection of stuck-at faults into the configuration memory of look-up tables.

The combinational logic in Field-Programmable Gate Arrays (FPGAs) is mostly implemented by means of Look-Up Tables (LUTs). Accordingly, any arithmetic/boolean expression of arbitrary complexity at RTL is mapped onto



```
entity parity is
 port (
  data : in std_logic_vector(7 downto 0);
  pbit : out std_logic );
end parity;

architecture behav of parity is
 function unary_xor (vect :std_logic_vector)
        return std_logic is
   variable res : std_logic;
   begin
    res := vect(vect'length-1);
    if vect'length > 1 then
      for i in (vect'length - 2) downto 0 loop
        res := res xor vect(i);
      end loop; end if;
    return res;
 end function;
begin
  pbit <= unary_xor(data);
end behav;
```

Fig. 6: Combinational logic: a) described at RTL, b) implemented by LUTs, and c) macrocell's internal structure

a set of interconnected LUTs at the implementation-level. As depicted in Fig. 6, each LUT consists of a tree of multiplexers controlled by the input address, which selects the output from the configuration memory cells.

From a robustness assessment perspective, this means that only input/output signals of arithmetic/boolean expressions are available for fault injection at RTL, whereas input/output ports of all LUTs can also be targeted by faults at the implementation level. This can be accomplished by the previously presented injection procedure for stuck-at, pulse, and indetermination faults. However, by considering the particular implementation of each macrocell, it is also possible to define specific fault injection approaches like, for instance, to study the sensitivity of configuration memory cells to bit-flip faults.

The logic function implemented by the *X_LUT6* macrocell, from Xilinx's SIMPRIM library [35], is defined as generic parameter (*INIT*). This parameter initialises an internal constant named *INIT_reg*, which represents the memory cells of the macrocell. This constant is used in the *Functionality* section of the *VITALBehavior* process to compute the expected output of the macrocell. Accordingly, the macrocell must be instrumented to use a signal instead of a constant and activate the process whenever this signal changes. The required instrumentation is deployed by the following operations: *constant2signal(INIT_reg); addToList(VITALBehavior, v_INIT_reg)*. After that, any bit of this truth table can be modified. For instance, a bit-flip targeting the bit 0 can be injected by means of *value=examine(v_INIT_reg); force(v_INIT_reg, freeze, value xor 0x00000001, 0).*

4 Robustness Assessment at RTL and Implementation Level: Embedded Processors as a Case Study

Three embedded processors with different architectures—32-bit SPARC V8 (LEON3) [5], Intel 8051 (MC8051) [21], and Microchip PIC16C5X (PIC) [23]—have been selected as a case study to illustrate the gap in robustness assessment between RTL and implementation level models. As the PIC and MC8051 cores

Table 2: Range of fault targets at RTL and implementation level for all the considered configurations of the toolchain parameters

Cores	RTL (signals)	Implementation level ([Minimum, Default, Maximum]) Flip-flops	Look-Up Tables
LEON3	1354	[905, 907, 986]	[1906, 2128, 3420]
MC8051	1287	[577, 608, 679]	[2310, 3218, 3728]
PIC	1070	[719, 719, 783]	[530, 695, 1327]

Table 3: Faultload Configuration

			Injections per target			
	Injection	Forced	RTL	Implementation		
Fault model	Time	Value	signals	Flip-Flops	LUTs	CMEM[1]
Bit-flip	Random	Inverted	3 / signal[2]	3 / FF[2]	–	–
Stuck-at-0	0	1	1 / signal[2]	1 / FF[2]	1 / LUT[2]	1 / bit
Stuck-at-1	0	0	1 / signal[2]	1 / FF[2]	1 / LUT[2]	1 / bit
Pulse	Random	Inverted	–	–	24 / LUT[3]	–
Propagation delay	0	$+\delta^4$	–	10 / path	10 / path	–
Interconnect delay	0	$+\delta^4$	–	10 / port	10 / port	–

[1] CMEM: Configuration memory cells of LUTs.
[2] At implementation level applied to 128 factorial configurations as well as to default configuration.
[3] Considering 3 instants in time with 8 different widths ($[0.02, 0.05, 0.1, 0.2, 0.3, 0.5, 0.8, 1.0] \times clk$)
[4] Delay increased by ($[0.02, 0.05, 0.1, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0] \times clk$)

will execute pre-compiled synthetic tests that activate most paths within the cores, and the LEON3 unit will run a matrix multiplication workload from the MiBench benchmark suite [11], robustness results cannot be compared among targets. It must be noted that differences between RTL and implementation-level models cannot be attributed to the workloads, as both models should behave exactly the same when running the same workload.

These models have been implemented using Xilinx's ISE Design Suite 14.7 for a Virtex-6 FPGA (6vcx240tff784-2), thus using Xilinx's SIMPRIM macrocells. The performance, area, power consumption, and robustness of the system depend on the actual configuration of the parameters of the toolchain. To take this variability into account, 128 configurations (plus the default configuration provided by the considered tools) have been implemented after [30], which ensures a statistically representative sample of all possible configurations. Table 2 lists the total number of fault injection targets, and shows a maximum difference of 12% sequential targets with respect to the default configuration and nearly 90% in the case of combinational targets.

The faultload applied to each model is listed in Table 3. Even though pulses, delays, and faults on the configuration memory of LUTs were injected just on the default configuration of the core to keep the experimentation time within reasonable limits, a total of 3.7 millions of experiments were carried out in a time frame of 15 days.

The structure and naming conventions of VITAL-compliant macrocells can be exploited to automatically build custom scripts to instrument selected macrocells and generate simulator commands sequences to inject a given fault.

DAVOS [32], a custom SBFI tool, has been extended to enable the injection of logic faults into SIMPRIM macrocells using the proposed generic operations.

The log files containing the raw measurements obtained for the default implementation of each microcontroller is publicly available at [31]. The outcome of an experiment will be classified as a *failure* when the outputs deviate from the expected ones. Thus, robustness is quantified as $100\% - failure\_rate$. The robustness assessment extracted form this logs is detailed in the next Section.

5 Analysis of Results

The experimentally estimated failure rate for each microcontroller in presence of the various injected faults is depicted in Fig. 7.

The diagram in Fig. 7a focuses on stuck-at-1/0 faults, as they can be injected into all available targets at both RTL and implementation level without distinguishing between sequential and combinational logic. As can be expected, results show that there exists a significant gap between the robustness estimated at RTL and implementation level, but what is more important is that the robustness of all three microcontrollers is overestimated at the RTL. Furthermore, this gap may become more pronounced depending on the particular configuration of the selected toolchain, as denoted by the error bars in Fig. 7. Since these fault models do not require precise timing information (post-place and route models) to obtain accurate robustness results, it is highly recommendable to run fault injection experiments as soon as synthesizable models are available, as post-map models already contain accurate information about the required combinational logic. In this way, it could be possible to detect existing dependability bottlenecks overlooked by the analysis at the RTL, and apply solutions that could be greatly costly to deploy after the implementation level is available.

The diagram in Fig. 7b shows that the estimated failure rate in presence of stuck-at-1/0 and bit-flips faults targeting the sequential logic of the system is very similar for both RTL and implementation level models. Thanks to the naming convention followed by Xilinx's toolchain it has been possible to trace each sequential macrocell back to the original RTL model. Thus, the same faults have been injected exactly at the same location in both models. Small differences observed with respect to the default implementation are attributable to the small percentage of targets that could not be traced back (1% for LEON3, 6% for MC8051, and 0% for PIC), due to the particular final implementation of the 128 different configurations. This result not only verifies the accuracy of the proposed fault injection procedure, but also enables the early and accurate injection of bit-flips into synthesizable HDL models, as soon as sequential signals can be traced back from a post-map model.

The diagram in Fig. 7c illustrates that, as expected, considering the whole macrocell as faulty underestimates the robustness of the system in presence of faults affecting its combinational logic. The very low failure rate for faults affecting the configuration memory representing the logic function of the LUTs

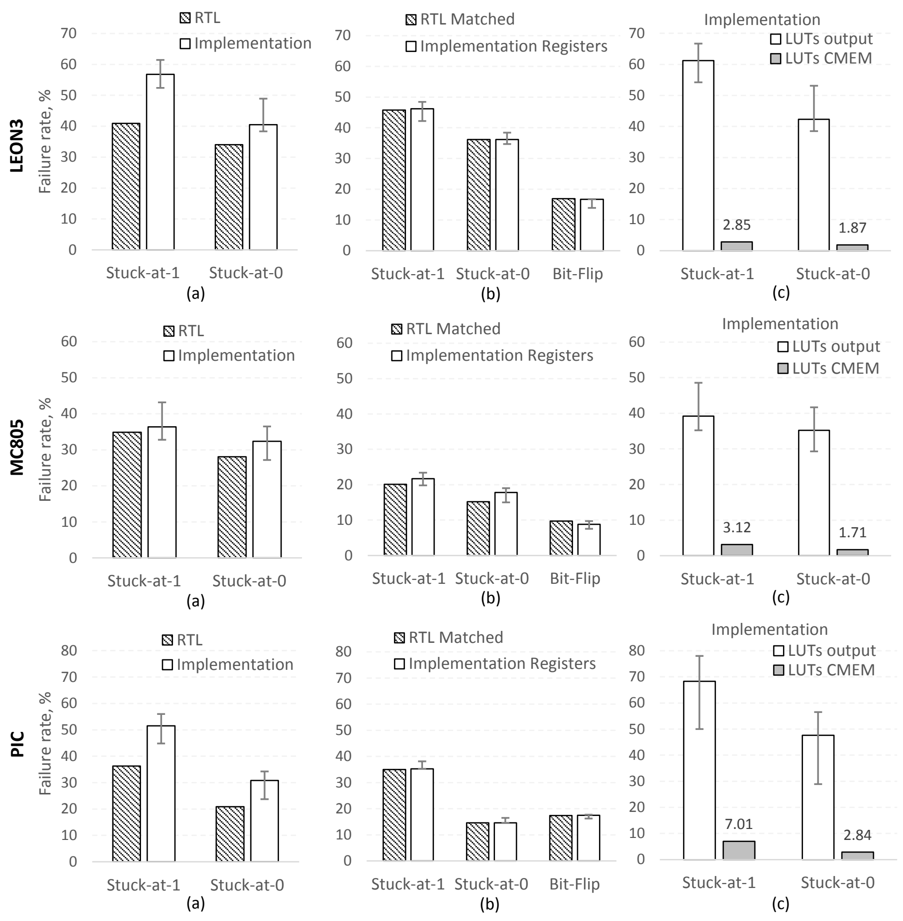


Fig. 7: Estimated failure rate for the default configuration in presence of stuck-at-1/0 and/or bit-flip faults targeting: (a) all signals at RTL and all macrocells at the implementation level, (b) all sequential macrocells at the implementation level and all signals matching them at RTL, and (c) the output or the configuration memory of all combinational macrocells at implementation level. Error bars represent the deviation observed for all considered implementations.

is due to the selected workload. Even though about 25% of the faults targeting active memory cells lead to failure, around the 80%, 44%, and 60% of cells were inactive for the LEON3, MC8051, and PIC, respectively. Accordingly, tailored procedures should be defined for each particular macrocell, so accurate and precise fault injection procedures could be deployed, although macroscopic approaches (macrocell is faulty) could also be used to get fast but inaccurate estimations. This inaccuracy greatly underestimates the robustness of the system, so costly fault tolerance mechanisms in terms of area, maximum clock frequency, and power consumption will be unnecessarily deployed.

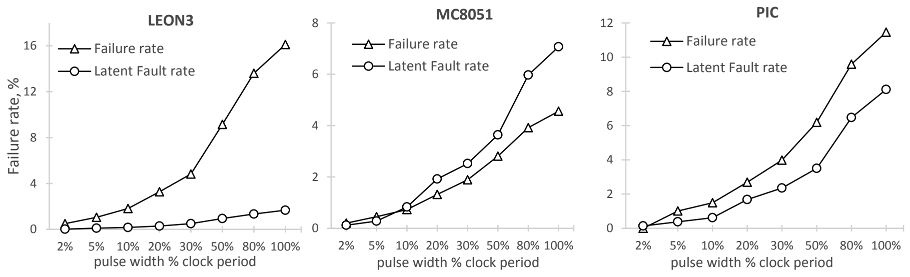


Fig. 8: Estimated failure rate for pulse faults with increasing width

Furthermore, implementation-level models provide accurate timing information that is required to consider delay faults and to estimate the effects of transient faults with respect to their duration.

For instance, Fig. 8 depicts the effects of pulse faults, targeting target combinational logic, with different widths. As can be expected, wider pulses lead to higher failure rates, but both factors are highly dependent on the target system. For instance, the width of the pulse must be of 50% and 80% of the clock period of the LEON3 and PIC (4.25 ns and 6 ns respectively) for the failure rate to reach a 10%, whereas it keeps below an 8% for the MC8051 with pulses as wide as the clock period (17 ns).

Special care should be taken when considering the propagation of pulses (or other transient faults), as they may be affected by the delay mode used within the macrocell. Four delay modes are defined in the VITAL standard. *VitalTransport* and *VitalInertial* behave like transport and inertial delays in VHDL: transport delay propagates the pulse independently from its width, whereas inertial delay filters out the pulses shorter than the specified propagation time. In both modes glitches are not detected. *OnEvent* and *OnDetect* modes enable glitch detection and, if the pulse is shorter than the propagation time, the undetermined *'X'* value is scheduled to the output. All examined components of the SIMPRIM library are defined with *VitalTransport* delay mode, therefore they do not affect the expected behavior of injected pulses. However, when working with different libraries, these modes should be taken into account for the accurate injection of pulses.

Finally, Fig. 9 illustrates the failure rate caused by interconnect and propagation delays of different magnitude. As can be seen, the impact of interconnect delays depend on the target macrocell's input. Delays on the *CLK* input of flip-flops already manifest as failures when its interconnect delay is increased a 5% of the clock period, and the failure rate reaches its maximum (35%) when the delay is increased a 40% of the clock period. Such a high failure rate can be explained by hold violations on *I* with respect to *CLK*. However, when the delay is increased a whole clock period, the flip-flop gets synchronized again with the rest of sequential elements (just missing one clock cycle), and the failure rate decreases to just 3%. Delays on the *I* input of flip-flops start to manifest as failures when its interconnect delay is increased a 60% of the clock

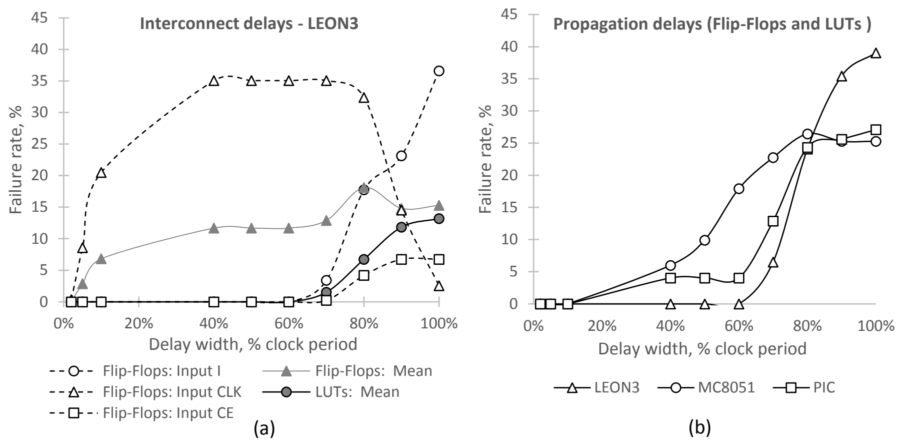


Fig. 9: Estimated failure rate for delay faults with increasing (a) interconnect and (b) propagation delays

period, but the failure rate grows very fast, reaching 36% when the delay is increased a whole clock period. Interconnect delays on the Clock Enable input *CE* of flip-flops have a much lower impact, reaching a maximum failure rate of 7% when the delay is increased by a whole clock period.

Regarding propagation delays, they start to manifest as failures for MC8051, PIC, and LEON3 when the delay is increased a 10%, 40%, and 60% of the clock period, respectively. For all of them, the failure rate grows rapidly when the propagation delay is increased between a 60% and 80% of the clock period.

6 Conclusions

The VITAL standard establishes a comprehensive set of rules to unify the design of macrocells libraries and EDA tools, enabling the implementation of efficient optimizations for simulation speed-up. However, such rigorous requirements make that common SBFI techniques cannot be easily applied to implementation level HDL models that use VITAL-compliant libraries.

This work has carefully studied the architecture of VITAL-compliant macrocells to define generic operations that can be deployed to enable the injection of the most common logic faults into implementation-level models. Sequences of generic simulator commands have been defined to conduct the injection of faults whenever possible, to reduce to the minimum the intrusion and the overhead on the simulation time. However, some faults can only be injected after instrumenting the target macrocells. In such cases, the defined operations keep the functionality and timing behaviour of the target macrocell while following VITAL requirements. Only in the case of interconnection path delays, the VITAL level of support will be degraded from 1 to 0. By instrumenting the macrocells, the original implementation-level model and the VITAL libraries are not modified by any means, reducing the intrusiveness of the proposed approach. Likewise, once the macrocell is instrumented and recomplied, no further recompilations are required, thus reducing also the experimental overhead with respect to other common approaches. The defined operations are generic enough to be technology independent, so they can be applied to the VITAL-compliant macrocells of any vendor and the commands can be supported by most common industry standard simulators. Obviously, this approach limits its applicability to VHDL-based models, whereas Verilog-based models are out of the scope of this contribution. Nevertheless, the same kind of problems appear when trying to inject logic faults at Verilog-based implementation-level models using simulator commands. In this case, the problem is that simulators use precompiled Verilog macrocells provided by vendors, so it is not straightforward to access and analyse the internal implementation of these macrocells. Further research is required to deal with this problem.

The proposed case study has considered the RTL models of three different microprocessors that have been implemented using the Xilinx's toolchain, and thus using the SIMPRIM library. Logic faults have been automatically injected by extending a custom SBFI to support the proposed unified specification. Results confirm the gap that exists between the assessed robustness at the RTL and the implementation level. However, if it is possible to match the signals representing the sequential logic at RTL with the macrocells obtained after mapping, then fast to simulate RTL models can be used for faults targeting the sequential logic of the system with an accuracy close to that of the final implementation. In case of faults targeting the combinational logic of the system, those dealing with timing information, or targeting device-specific components, it is mandatory to rely on implementation level models.

Our future work will focus on extending the proposed specification and set of logic fault models to consider routing-related faults and Verilog-based macrocells. As accurate robustness assessment can only be obtained at the implementation level for these faults, we will also continue to devise new techniques to speed-up the SBFI process.

References

1. Baraza, J.C., Gracia, J., Blanc, S., Gil, D., Gil, P.: Enhancement of fault injection techniques based on the modification of vhdl code. In: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 16, pp. 693–706 (2008)
2. Baraza, J.C., Gracia, J., Gil, D., Gil, P.: A prototype of a vhdl-based fault injection tool: description and application. Journal of Systems Architecture **47**(10), 847–867 (2002)
3. Benites, L.A.C., Kastensmidt, F.L.: Fault injection methodology for single event effects on clock-gated asics. In: IEEE Latin American Test Symposium, pp. 1–4. IEEE (2017)
4. Benso, A., Prinetto, P.: Fault Injection Techniques and Tools for VLSI reliability evaluation. Frontiers In Electronic Testing. Kluwer Academic Publishers (2003)
5. Cobham Gaisler AB: LEON3 processor product sheet (2016). URL `http://www.gaisler.com/doc/leon3_product_sheet.pdf`
6. Cohen, B.: VHDL Coding Styles and Methodologies. Springer US (2012)
7. Das, S.R., Mukherjee, S., Petriu, E.M., Assaf, M.H., Sahinoglu, M., Jone, W.B.: An improved fault simulation approach based on verilog with application to iscas benchmark

circuits. In: IEEE Instrumentation and Measurement Technology Conference, pp. 1902–1907 (2006)
8. Fernandez, V., Sanchez, P., Garcia, M., Villar, E.: Fault Modeling and Injection in VITAL Descriptions. In: Third Annual Atlantic Test Workshop, pp. o1–o4 (1994)
9. Gil, D., Gracia, J., Baraza, J.C., Gil, P.: Study, comparison and application of different vhdl-based fault injection techniques for the experimental validation of a fault-tolerant system. Journal of Systems Architecture **34**(1), 41–51 (2003)
10. Gil, P., Arlat, J., Madeira, H., Crouzet, Y., Jarboui, T., Kanoun, K., Marteau, T., Duraes, J., Vieira, M., Gil, D., Baraza, J.C., Gracia, J.: Fault Representativeness. Tech. rep., Dependability Benchmarking project (2002)
11. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization, pp. 3–14 (2001)
12. IEEE Standard for VITAL ASIC (Application Specific Integrated Circuit) Modeling Specification. Standard, Institute of Electrical and Electronic Engineers (2000)
13. IEEE Standard VHDL Language Reference Manual. Standard, Institute of Electrical and Electronic Engineers (2008)
14. Jenn, E., Arlat, J., Rimen, M., Ohlsson, J., Karlsson, J.: Fault injection into vhdl models: the mefisto tool. In: International Symposium on Fault-Tolerant Computing, pp. 66–75 (1994)
15. Kochte, M.A., Schaal, M., Wunderlich, H.J., Zoellin, C.G.: Efficient fault simulation on many-core processors. In: Design Automation Conference, pp. 380–385 (2010)
16. Mansour, W., Velazco, R.: An automated seu fault-injection method and tool for hdl-based designs. IEEE Transactions on Nuclear Science **60**(4), 2728–2733 (2013)
17. Mentor Graphics: Questa SIM Command Reference Manual 10.7b, Document Revision 3.5 (2016). URL `https://www.mentor.com/products/fv/modelsim/`
18. Munden, R.: ASIC and FPGA Verification: A Guide to Component Modeling. Systems on Silicon. Elsevier Science (2004)
19. Na, J., Lee, D.: Simulated fault injection using simulator modification technique. ETRI Journal **33**(1), 50–59 (2011)
20. Nimara, S., Amaricai, A., Popa, M.: Sub-threshold cmos circuits reliability assessment using simulated fault injection based on simulator commands. In: IEEE International Symposium on Applied Computational Intelligence and Informatics, pp. 101–104 (2015)
21. Oregano Systems GmbH: MC8051 IP Core, User Guide (V 1.2), 2013 (2013). URL `http://www.oreganosystems.at/download/mc8051_ug.pdf`
22. R. Munden: Inverter, STDN library, Free Model Foundry VHDL Model List (2000). URL `https://freemodelfoundry.com/fmf_models/stnd/std04.vhd`
23. Romani, E.: Structural PIC165X microcontroller. Hamburg VHDL Archive (1998). URL `https://tams-www.informatik.uni-hamburg.de/vhdl`
24. IEEE Standard for Standard Delay Format (SDF) for the Electronic Design Process. Standard, Institute of Electrical and Electronic Engineers (2001)
25. Shaw, D., Al-Khalili, D., Rozon, C.: Automatic generation of defect injectable vhdl fault models for ASIC standard cell libraries. Integration, the VLSI Journal **39**(4), 382–406 (2006)
26. Shaw, D.B., Al-Khalili, D., et al.: Ic bridge fault modeling for ip blocks using neural network-based vhdl saboteurs. IEEE Transactions on Computers (10), 1285–1297 (2003)
27. Short, K.L.: VHDL for Engineers, 1 edn. Pearson (2008)
28. Sieh, V., Tschache, O., Balbach, F.: Verify: Evaluation of reliability using vhdl-models with embedded fault descriptions. In: International Symposium on Fault-Tolerant Computing, pp. 32–36 (1997)
29. Singh, L., Drucker, L.: Advanced Verification Techniques. Frontiers In Electronic Testing. Springer US (2004)
30. Tuzov, I., de Andrés, D., Ruiz, J.C.: Dependability-aware design space exploration for optimal synthesis parameters tuning. In: IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 1–12 (2017)
31. Tuzov, I., de Andrés, D., Ruiz, J.C.: Robustness assessment via simulation-based fault injection of the implementation level models of the LEON3, MC8051, and PIC microcontrollers in presence of stuck-at, bit-flip, pulse, and delay fault models [Data set]. Zenodo (2017). URL `http://doi.org/10.5281/zenodo.891316`

32. Tuzov, I., de Andrés, D., Ruiz, J.C.: DAVOS: EDA toolkit for dependability assessment, verification, optimization and selection of hardware models. In: IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 322–329 (2018)
33. Tuzov, I., Ruiz, J.C., de Andrés, D.: Accurately Simulating the Effects of Faults in VHDL Models Described at the Implementation-Level. In: European Dependable Computing Conference, pp. 10–17 (2017)
34. Wang, L.T., Chang, Y.W., Cheng, K.T.: Electronic Design Automation: Synthesis, Verification, and Test. Morgan Kaufmann (2009)
35. Xilinx: Synthesis and Simulation Design Guide, UG626 (v14.4) (2012). URL `https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/sim.pdf`