Fibich *et al. EURASIP Journal on Embedded Systems* (2019) 2019:2
https://doi.org/10.1186/s13639-019-0088-7

EURASIP Journal on Embedded Systems

RESEARCH Open Access

FIJI: Fault InJection Instrumenter

Christian Fibich[1*], Stefan Tauner[1], Peter Rössler[1], Martin Horauer[1], Martin Matschnig[2] and Herbert Taucher[2]

Abstract

FPGAs are increasingly used in safety-critical applications (e.g., in aerospace and automotive engineering). Safety standards stipulate that implemented countermeasures against run-time faults such as detection and isolation of affected components, automatic reconfiguration, and redundancy mechanisms must be adequately verified. To that end, fault injection tests by various means have been established as a suitable method.
For such tests, faults can be provoked by radiation, simulation, or manipulating the design, for example, by inserting additional logic or manipulating the synthesis flow. This work briefly summarizes the various fault injection approaches with a focus on methods that are capable of stressing critical nets of a design running on actual hardware without requiring to re-synthesize. While the state-of-the-art tools can work with complex designs, they often lack controllability of the exact timing of the injection events (which is important to track the system's response on faults in a logic simulation) and/or use a high amount of FPGA resources. To overcome these issues, we propose a resource-saving netlist-based fault injection framework *Fault InJection Instrumenter (FIJI)* that can target individual nets at test runtime. This paper presents FIJI's work flow, implementation details, and an evaluation in terms of FPGA resources, timing impact, and performance during instrumentation and test execution. The FIJI framework has been made publicly available by the authors under an open-source license.

Keywords: Fault injection, FPGA, Safety-critical system, Verification, Electronic design automation

1 Introduction

FPGAs are increasingly used in safety-critical systems such as (partially) autonomous cars, airplanes, spacecrafts, and industrial applications. For the respective applications, they offer several advantages to possible alternatives such as ASICs or general purpose processors. The most distinctive ones are customizability, short time to market, and (fast) run-time reconfiguration [1].

However, due to their architecture, the usual harsh environmental conditions they operate in, and strict (legal) requirements on high availability and safety, many aspects need to be considered before using them in such applications. A survey on the legal and methodological standards by Bernardeschi et al. [2] concludes: "The main issues related to the design of FPGA-based systems and their adoption in safety-critical application fields are the lack of standards specifically addressing the FPGA technology and the severe susceptibility of FPGA devices to the effects of radiations". However, the frequent use of FPGAs in other sectors and their unique features increases the pressure on regulatory bodies and manufacturers alike to consider applicability of standards to reconfigurable logic in general and FPGAs in particular. For example, the European Cooperation for Space Standardization (ECSS) (responsible for standards mandatory for contractors working for ESA) published a set of requirements for FPGA development in space projects [3]. FPGAs have also widely been used in projects subject to the DO-254 aviation standard [4].

Today, the predominant type of FPGAs rely on SRAM to hold their configuration during operation. This configuration specifies not only the behavior of functional blocks within the FPGA but also their interconnections. Thus, it is of utmost importance to retain the content of this memory for correct system behavior. Due to the vulnerability of SRAM cells to radiation-induced single event upsets (SEUs), FPGAs using other types of memory for configuration (e.g., flash or antifuses) might come to mind that do not exhibit the same vulnerability. However, due to the advantages of fast reconfigurability, there

*Correspondence: fibich@technikum-wien.at
[1]University of Applied Sciences Technikum Wien, Höchstädtplatz 6, 1200 Vienna, Austria
Full list of author information is available at the end of the article

is much interest in using SRAM-based devices instead. Also, flash-based devices come with their own set of problems that make them unsuitable for long-term space missions. To asses the fault coverage as mandated by many of the standards and to evaluate possible countermeasures, fault injection (FI) tests have been established as a suitable method where analytical approaches are too complex [5]. During FI tests, the normal behavior of a system is changed by irritating some of its parts by different means.

The remainder of this section gives an overview of previous research in this context. This includes FI by (i) Using radiation to induce upsets in actual hardware, (ii) Simulating errors in the configuration memory of FPGAs, (iii) Attempting to modify the FPGA's configuration bitstream file directly, and (iv) Modifying the FPGA's configuration during operation using some run-time reconfiguration methods. A review of scientific work that deals with injecting arbitrary errors in a more directed manner follows in Section 2.

Due to SRAM's susceptibility to SEUs, retention of the configuration memory's contents in the presence of (different types of) radiation cannot be taken for granted at all. Extensive research has been conducted in the past on the effects of radiation, reproducing them for testing as well as effective mitigation techniques [5]. The respective experiments are very costly and can only give probabilistic results due to their rather unfocused nature.

Simulating the errors occurring within an FPGA device on different levels of abstractions poses a cheap and more directed alternative to physically inducing them. Bernardeschi et al. present a simulation framework that improves the accuracy of simulated errors in the configuration memories over traditional stuck-at fault models and models emulating logic components only without taking the interconnect into account [6]. To assess the impacts of SEUs in the memory specifying the routing within the FPGA, they derive and exploit information on configuration memory of Xilinx FPGAs. This has the advantage of directly disturbing the in-device description of the circuit function and great observability (e.g., possibility to determine affected circuit elements and resulting fault modes). However, this requires detailed knowledge about the FPGA architecture and as simulation techniques usually require impractical amounts of run time with complex designs.

Somewhat less specifics need to be known when targeting the bitstream of the devices to introduce random faults. The bitstream is the representation of the FPGA's configuration for a distinct application in the form of a file that is downloaded to the device before/on startup. The exact semantics of bitstreams are proprietary vendor secrets that have increased in complexity over the years. If enough details are known to modify them, FI can be applied by modifying the bitstream as output by the vendor's implementation tool. With this technique, Asadi et al. [7] perform evaluation of the effects of configuration defects in Altera Flex10K SRAM-based FPGA devices by generating one faulty bitstream file for each targeted FPGA resource. The resulting designs are then consecutively downloaded to the FPGA and subjected to the same test patterns as a fault-free bitstream to determine any effects on the behavior/outputs of the design.

Such targeted FI into a generated bitstream file is only possible if the vendor's bitstream file format is available. For modern devices of the two largest FPGA vendors, Intel (Altera) and Xilinx, almost no information on their bitstream format is publicly available. A fault injection tool targeting the bitstream file would have to support the specifics of each distinct FPGA device to be used and be able to produce correct checksums if need be. Moreover, manipulating the FPGA's bitstream allows insertion of permanent faults only.

For these reasons, a large number of FI approaches for FPGA hardware makes use of the ability of modern FPGA to access the configuration memory at runtime. Approaches from academia as described by Sterpone and Violante [8], Legat, Biasizzo and Novak [9], Straka, Kastil and Kotasek [10], Mogollon et al. [11], and Azkarate-askasua et al. [12] mainly target Xilinx FPGAs due to the availability of the relatively open Internal Configuration Access Port (ICAP) and external SelectMAP/JTAG interfaces which make it possible to read, modify, and write back configuration frames.

The FI system described by Sterpone and Violante [8] is controlled by a hard-core processor (PowerPC), which communicates with a host workstation over a serial interface. Fault patterns are stored in an on-chip memory and subsequently applied via ICAP. A very similar approach is taken by Legat, Biasizzo, and Novak [9], who use a soft-core MicroBlaze CPU and off-chip memory instead.

Straka, Kastil, and Kotasek [10] use the host PC directly as FI generator via the FPGA's JTAG port by exploiting Xilinx ChipScope's TCL bindings. The design is duplicated to act as a golden sample on the one hand while its configuration in the second instance is modified to simulate faults on the other hand. The comparison of outputs between these instances is done directly on-chip, and results are exported via a UART.

FTUNSHADES2, described by Mogollon et al. [11], is an even more elaborate FI test system that uses two dedicated FPGAs to generate the stimuli and evaluate results respectively. The system consists of a motherboard hosting a *Control FPGA* and two or more daughterboards connected via PCIe, each hosting a target *T-FPGA* and a stimuli-generating *S-FPGA*. In FPGA testing mode, one

of the daughterboard's FPGA is configured with a golden, fault-free bitstream, while the other FPGAs are configured with faulty bitstreams. Stimuli are applied by the *S-FPGA* and the supervising control FPGA configures them via SelectMAP and compares their outputs.

Azkarate-askasua et al. [12] focus their work on injecting faults into complex multi-core Network on Chip (NoC) designs implemented in Xilinx FPGAs. The general approach is similar to that of Legat [9] as they also use the ICAP interface to change the FPGA's configuration at runtime and control the process with a soft-core MicroBlaze CPU. Their monitor sits on the same NoC connecting the independent cores thus greatly reducing the load of the monitor by exploiting the inherent fault tolerant features.

Lately, both Xilinx and Intel (Altera) provide functionality to perform FI tests with less effort:

Xilinx's Vivado Design Suite includes an IP core referred to as *Soft Error Mitigation (SEM) Core* [13] that is primarily meant to automatically detect and correct any errors in the configuration data during runtime but is also able to perform FI on the FPGA's configuration. Each injection is toggling a single bit in the configuration memory that can be selected by the user via a logical (linear) or physical address. Neither is clearly specified by the vendor to easily allow the user to select the logic targeted by the error. Additionally, the SEM IP does not provide exact control over the timing of the FI operations.

Altera provides the *Fault Injection Debugger* [14] as a component of its Quartus II implementation tool since version 14.0. It is similar in function and scope to Xilinx' SEM core. One difference is that the location of injections can be limited to single-design partitions, which form logical boundaries in design hierarchies. Since the granularity of design partitions is at the level of HDL entities it is not possible to target individual nets with this mechanism.

In summary, information generally not disclosed by the FPGA vendors is necessary for targeted FI using simulated or actual errors in the FPGA's configuration memory. To work around this some researches rely on cumbersome and inaccurate methods to map between addresses and design elements [15], or tedious reverse engineering of the internal structure of configuration frames [9]. If this information is not present at all, at best only random tests can be carried out, making duplication of error conditions virtually impossible if the design changes. Even if this fact is accepted, the reconfiguration-based FI method remains highly vendor-dependent.

While probabilistic assessments on fault tolerance as described above are certainly an important part of the overall evaluation of any safety critical system it might not be sufficient for some steps in the design process. When implementing and validating features that are meant to improve the safety of an HDL design, it is necessary to manipulate the state of individual nets which is not possible with the methods discussed so far. These injections not only require to target arbitrary nets but might need to synchronize faults precisely with the behavior of the remaining design.

In the next section, alternative FI methods are discussed that are capable of being locally and temporally constrained. After demonstrating their downsides, we introduce our FI framework in Section 3, followed by implementation details of some distinct parts of the framework in Section 4. An analysis of key performance indicators of the framework as well as a comparison with its strongest competitor can be found in Section 5. At the end, in Section 6, a summary of the paper and an outlook on further work is given.

2 Related work

This section elaborates on FI techniques that can inject into arbitrary nets in an HDL design and/or can be precisely timed in relation to the design under test (DUT) (which is not possible with bitstream-related approaches, see Section 1).

Existing academic literature in this area modify either (i) The design's HDL description on RTL, or (ii) The design's post-synthesis netlist.

2.1 HDL-level approaches

Approaches that modify the DUT's HDL representation are performed by Baraza et al. [16], Grinschgl et al. [17], and Jeitler, Delvai, and Reichör (FuSE) [18]. Baraza et al. describe two main ways of injecting faults into an HDL description: (i) *Saboteurs* are distinct HDL modules that are instantiated in the targeted design unit to manipulate individual signals. These modules are capable of changing the original value of the signal to emulate the presence of a fault. (ii) *Mutants*, on the other hand, are created by modifying the functionality implemented in the HDL description of the targeted design unit itself, e.g., by modifying signal assignments or conditions of `if` statements.

Baraza et al. address simulation only, while Jeitler, Delvai, and Reichör target a specific simulation accelerator with dedicated hardware.

Grinschgl et al. use the *saboteur* approach only but focus on actual hardware. Here, the FI process is controlled by a distinct design unit referred to as *fault injection controller*, which makes use of a hard-core CPU embedded in the target FPGA. In their case study, Grinschgl et al. inject faults into a LEON3 CPU. Neither addresses the problem of accurately timing FI events.

2.2 Netlist-level approaches

Zheng, Fan, and Yue (*FITVS*) [19] as well as Mansour and Velazco (*NETFI*) [20], and Pellegrini et al. (*CrashTest*) [21] implement FI at the netlist level. All share the basic approach of synthesizing the DUT description first into Verilog or EDIF netlists before instrumenting these and/or the underlying technology-independent synthesis libraries with custom tools. The result contains wrappers for the original primitives providing additional combinational logic and inputs to emulate faults. Additionally, the control inputs and result outputs for the FI have to be integrated and forwarded through the hierarchy.

Zheng, Fan, and Yue use a custom *fault emulation controller* within the FPGA to download FI data from a host workstation and orchestrate the FI process, while Mansour and Velazco use a hard-core PowerPC CPU embedded in a distinct *Controller FPGA* and some controlling logic in the *DUT-FPGA* for this task. The additional FPGA was later replaced by a soft-core MicroBlaze CPU synthesized into the same device as the DUT in the *NETFI-2* flow [22].

In their *CrashTest* flow, Pellegrini et al. first synthesize the original HDL design into a Verilog gate-level netlist using the GTECH library of Synopsys Design Compiler (DC). This netlist is then parsed by a Perl script that automatically selects nets to be targeted by FI. These nets are then broken up, and FI logic emulating either stuck-at, stuck-open, bridging, delay, or transient faults is inserted. The respective fault model is selected at instrumentation time, and only one fault model can be implemented per instrumented netlist. Register chains are added for configuring and activating the FI logic at runtime. However, the activation of faults happens asynchronously to the DUT in software of an attached CPU thus no accurate timing is possible. Controlling the FI process was originally done by a hard-core PowerPC CPU in a Xilinx Virtex-2 but eventually was implemented in a soft-core MicroBlaze CPU providing a serial terminal to a host workstation [23]. In the published case studies [21], faults are injected into a Sparc (LEON3) and a MIPS-like (DLX) CPU, as well as an NoC router core. Additionally, there is also a use case available online targeting an unmodified version of Sun's OpenSPARC implementation. The source code for both the instrumentation tool and the HDL designs are publicly available from the authors.

2.3 Discussion

While less problematic with respect to vendor dependence and available information than the approaches in Section 1, one possible drawback of FI at RTL is that real-world designs often include components described in both VHDL and Verilog. A real-world FI tool thus must be able to parse and generate code in both languages, which increases complexity and thus the likelihood for bugs in the software that modifies the HDL. Moreover, "sabotaged" and "mutated" HDL may result in entirely different logic after synthesis than the original design due to different logic optimizations being applied, including retiming and resource sharing. Thus, it is difficult to ensure that the faults are actually inserted at the intended hardware location and have no unforeseen consequences on the DUT.

FI at the netlist level seems to be the most reasonable approach for injecting faults in a running FPGA design: (i) The process becomes reasonably vendor-independent with the usage of a distinct logic synthesis tool such as Synopsys' Synplify or Mentor Graphics' Precision. (ii) It does not require any non-disclosed knowledge about the targeted FPGA device such as bitstream format or location of resources in the configuration memory. (iii) It provides reasonable accuracy of the location of the injected fault, as injection is done in an already optimized post-synthesis netlist.

Arguably, the only drawback of this method is that some signals present at the HDL level may not be present anymore in the post-synthesis netlist.

Approaches like *FITVS* [19] and *NETFI* [20] that modify the FPGA primitives library are problematic because this modification has to be done for each FPGA family and vendor to be supported. Moreover, no separation between FI logic and the DUT's logic is possible as the output is only one fault-injected netlist. Finally, it is often not necessary to inject faults in each and every cell present in an FPGA design potentially increasing the resource usage beyond applicability.

Although *CrashTest* is publicly available and uses a netlist-based flow, it has the following drawbacks: (i) The dependence on Synopsys' DC instead of a less expensive FPGA synthesis tool such as Synopsys' Synplify or Mentor Graphics' Precision that generate netlists in a standardized format (e.g., EDIF or Verilog), (ii) The impossibility to trigger different fault models in a single netlist or switch between them at runtime, (iii) The lack of a control over the FI timing—making simulation or even reproduction of the same error condition impossible, and (iv) Most importantly, the high hardware consumption caused by the MicroBlaze-based FI controller, which is (v) Available for Xilinx FPGAs only.

Hardware overhead incurred by the instrumentation of a design with FI logic is a critical factor [24]: A FI system with high hardware overhead may not fit into the target FPGA together with the design under test. Proving that test results obtained with a larger FPGA device are equally valid in the (smaller) target FPGA may be difficult, as a different device may lead to changed implementation

results. This virtually forbids the usage of full-scale soft-core processors for controlling FI in the context of fault-tolerant designs subject to regulations for certification. Using hard-core CPUs is also problematic as the DUT most likely already depends on these resources for its main application.

The ability to control the exact (cycle-accurate) timing of the injected faults is also an important feature of a FI tool in order to track the system's response on faults in a logic simulation. Figure 1 shows an example: A state machine observes signal `s_activate` in State `ACT_2` and moves either to State `IDLE` (if `s_activate` = 0) or State `ACT_3` (if `s_activate` = 1) respectively. The dotted line of s_activate shows the case when an SEU fault is injected during State `ACT_2`, that inverts `s_activate` for one clock cycle and, thus, causes the state machine to move on to State `IDLE`. If the timing of the SEU fault cannot be reproduced cycle-accurately in a logic simulation (`s_activate` must be set to logic 0 during State `ACT_2`) the behavior that is seen during simulation would be different (the state machine would move on to State `ACT_3`). As already mentioned in Section 1, bitstream-related FI approaches based on, e.g., [13] and [14], do not provide exact control over the timing of the FI operations. Therefore, it would be difficult to reproduce fault scenarios in a simulation using these approaches.

3 The FIJI framework

As all of the previously existing FI solutions described in the academic state of the art either require access to the configuration memory format, rely on expensive tools, or are resource intensive through the use of either an entire hard-core or a soft-core CPU (see [24] and Section 5.4), the decision was made to develop a FI framework to improve on all these disadvantages.

In this work, we present *Fault InJection Instrumenter (FIJI)*, an open-source netlist-level FI framework for FPGAs. [25, 26] FIJI provides tools to select nets from a structural Verilog netlist, to instrument these nets with FI logic (i.e., *saboteurs*), and to execute FI tests. The scope of this framework is to evaluate the effects of faults on selected nets on the overall behavior and functionality of (usually fault-tolerant) designs. The following subsections give an overview of the architecture, tool flow, and some distinct features of FIJI.

3.1 Fault injection hardware architecture

The FI logic that is part of FIJI is provided as a parameterizable HDL design. Due to the absence of any technology-dependent components such as BRAMs, PLLs/clock managers, or hard CPU macros, the FI hardware can be implemented on any FPGA device. Additionally, the FI hardware was implemented as resource-saving as possible and does not make use of complex hardware such as soft CPU cores.

Figure 2 depicts a complete FIJI system comprising the DUT (bottom right), together with the FIJI logic (top right) in an FPGA connected to a controlling host computer (left).

FIJI works by instrumenting individual nets of a given netlist of a DUT with FI logic according to a predefined FI configuration. For each affected net there exists exactly one Fault Injection Unit (FIU) that can alter the respective signal. A single fault injection controller (FIC) manages these FIUs and other components of the FIJI logic. It can be parametrized to fit the original design and required test properties for low resource usage. The FIC is connected to a host computer via a UART unit to allow remote changes of the FIU configurations and control the execution of the system.

FIJI automates most of the required steps to create such a system, including the generation of a top-level design that hosts FIJI's FI logic and the instrumented DUT, as explained in the next subsection.

3.2 Netlist transformation and execution flow

This section briefly describes the steps required from a given user HDL design to the execution of FI experiments

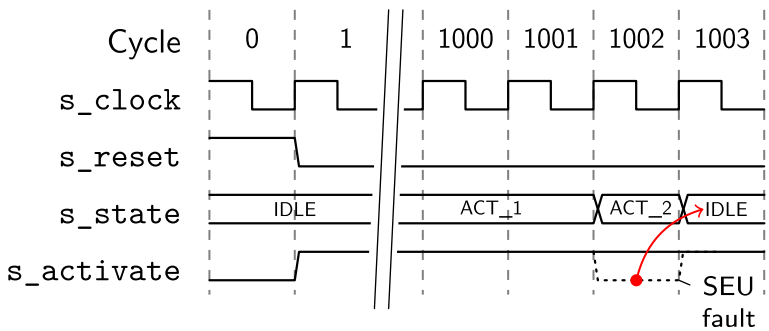

Fig. 1 Example illustrating the importance of exact FI timing

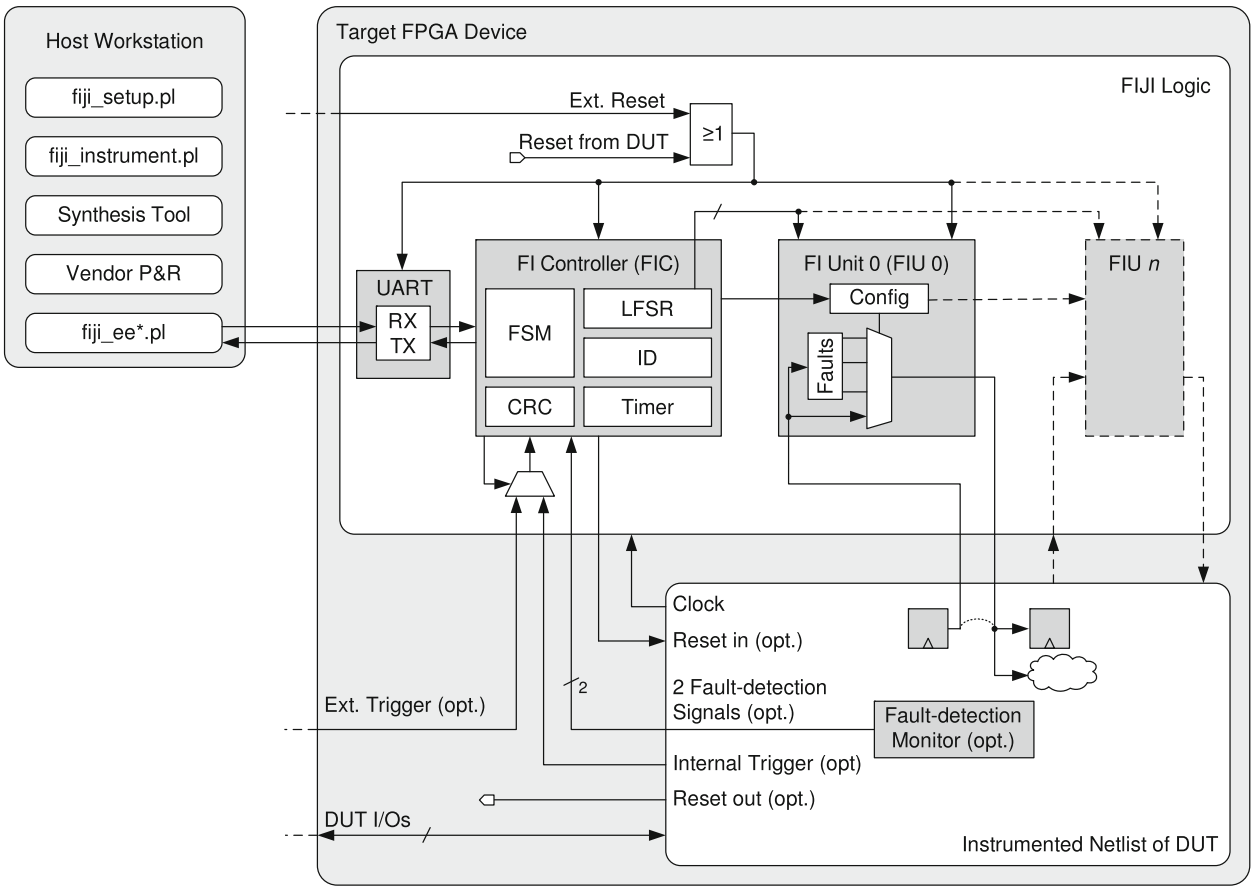

Fig. 2 Hardware overview (DUT and FI logic)

of an instrumented version of that design. An overview of the tool flow implemented by FIJI can be seen in Fig. 3.

The flow starts with the creation of an *original Verilog netlist* of the design that can be obtained using a logic synthesis tool from the original HDL design. In the second step, the user configures various aspects of the instrumentation via FIJI's *Setup* tool. This tool allows the user to enter nets to be targeted and options of the FI hardware subsystem such as interaction points with the DUT for resetting, timing, and triggering.

FIJI Setup generates a configuration file that is to be forwarded to the *instrumentation* tool together with the unmodified *original netlist*. The instrumentation tool parses this netlist, alters it by inserting *saboteurs* at selected places according to the configuration, and embeds the now instrumented DUT design into a generated wrapper containing the other components of the FIJI hardware and their interconnect. The wrapper has essentially the same input and output ports as the *original netlist* (apart from some additional pins), allowing to re-use existing pin constraints.

The user is then required to synthesize and implement (i.e., place and route) this wrapper to produce a *bitstream*. Some constraints that may be needed to implement the design (e.g., clock constraints, pin locations) have to be ported manually from the original design due to this hierarchy change. Once a device is configured with this bitstream via a *download tool*, and the relevant FIJI ports connected to a host PC, the design is ready for FI campaigns.

Finally, FIJI provides an *execution engine* that communicates with the generated FI logic over a serial interface and lets the user direct the FI operations. Faults can be injected either in a one-shot manner with manually set parameters, as a pre-configured sequence, or in a constrained random sequence. After a test run, the *execution engine* allows to export simulation templates to recreate the executed FI run in either RTL or gate-level simulation.

3.3 Fault models

FIJI supports the injection of faults according to four different fault models. The intent in the selection of these

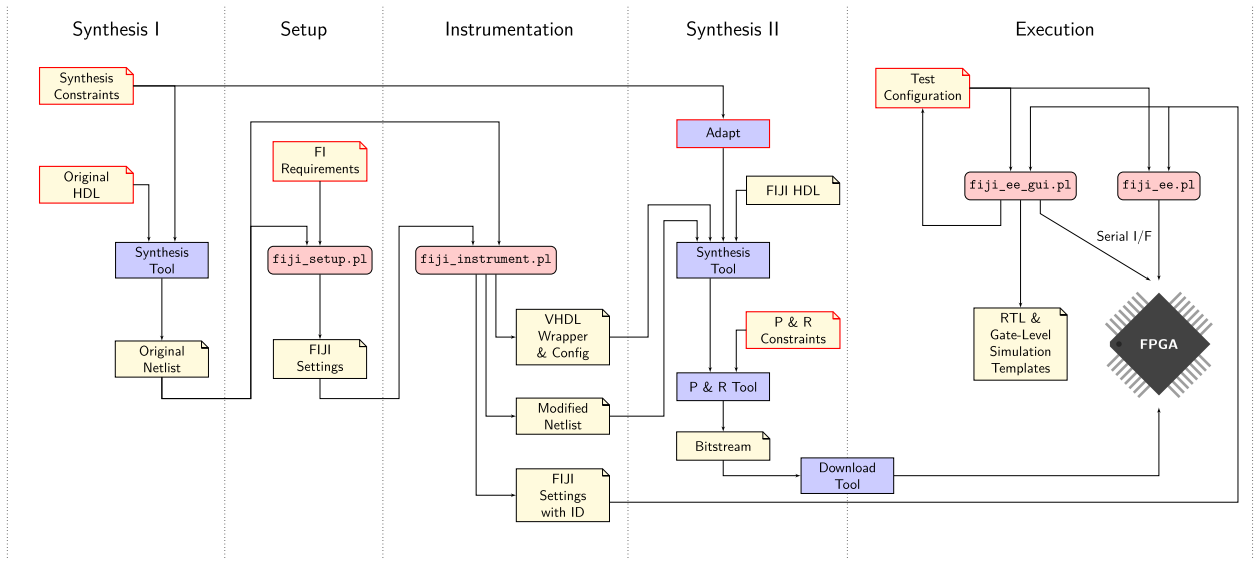

Fig. 3 FIJI tool flow

models was to be able to emulate a large share of the low-level fault modes to be expected in an integrated circuit either due to aging effects or due to soft errors. The selection of the fault models has also been inspired by the *CrashTest* framework [21] but omits its bridging fault model. Adding this and other theoretically useful fault models was taken into account when designing and implementing FIJI and thus should be straightforward if need be.

The fault models supported by FIJI are explained below and visualized in Fig. 4:

- Stuck-at-faults (0/1) allow to model a large share of faults that are experienced in an integrated circuit, e.g., faults in LUTs.
- Stuck-open allows to model a floating net, e.g., due to an interconnect fault by toggling the net randomly.

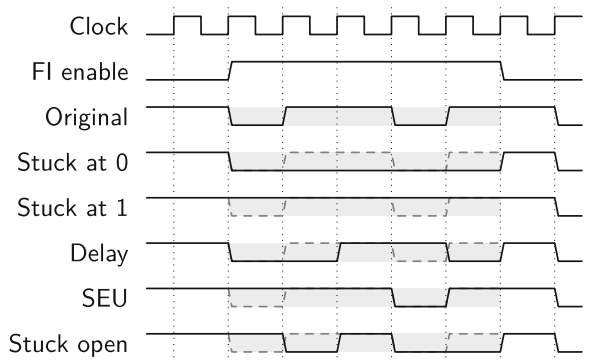

Fig. 4 Timing diagrams of fault modes

This is done by connecting the net to the output of an Linear Feedback Shift Register (LFSR).
- Delay allows to model faults that arise due to timing violations, e.g., due to temperature or capacitance effects.
- The SEU fault model allows to invert an instrumented net for one clock cycle. The aim of this fault model is to emulate the effect of a particle strike.

3.4 Fault injection timing

FIJI allows selecting the desired fault model at runtime. This selection, as well as the timing and other control capabilities described in the remainder of this section, are controlled by a host PC via an external serial interface. The protocol that is exchanged over this interface is message-based; one FI message contains two fault models for each FIU to be applied subsequently.

In order to be able to emulate faults of precise duration and frequency, FIJI provides capabilities to time the application of the selected fault models with clock-cycle accuracy. This is especially important when attempting to recreate a fault condition found using FIJI in simulation later on.

Timing of the injected faults is implemented entirely in the FI hardware subsystem, with a hardware timer of configurable width. As stated earlier, FIJI allows to configure two fault patterns per configuration message that are applied in two different phases of the FI whose duration is influenced by the hardware timer. An overview of a single FI sequence can be seen in Fig. 5.

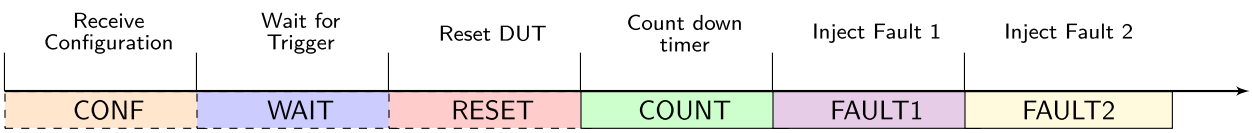

Fig. 5 Overview of FI phases

Initially, when no FI message has been received by the FI hardware, all FIUs are configured as pass-through, i.e, they do not influence the original signal.

The sequence begins when the first configuration is downloaded in phase *CONF*. After successful reception, the hardware commences an optional *WAIT* phase if instructed to do so by the configuration message. The hardware can wait for an edge on an internal or external trigger signal. This allows for precise synchronization of injection with stimuli generated by external hardware or signals of the DUT.

The following phase *RESET* is optional (and selectable via the FI message) as well. If enabled, an output signal is activated that can be used to reset (parts of) the DUT. Its duration in clock cycles and its level can be set at configuration time.

Phase *COUNT* is the last phase that merely changes the timing behavior. Its duration is determined by the first of two timer values transmitted per configuration. Afterwards, the FIUs are instructed to apply the first fault model. The first timer value t_1 thus specifies the duration to wait after the reception of the message, the potential registering of a trigger edge, and the potential application of the internal reset signal before applying the first fault model.

In the *FAULT1* phase, the second timer value t_2 is counted down. Afterwards, the second downloaded fault model is applied by the FIUs during *FAULT2*. The second timer value t_2 thus specifies the number of clock cycles the first fault pattern is applied by the FI logic. During the *FAULT1* phase, the FI hardware is ready to receive the next fault configuration message consisting of the two next timer values and fault patterns. If this message was received before *FAULT1* phase was completed, the cycle restarts with the optional *WAIT* phase as soon as *FAULT1* ends. The second fault pattern thus stays active until the timer value t_1 of the next configuration has been counted down. This overlapping of FI operations and the reception of the next fault pattern allows the seamless back-to-back injection of faults, further facilitating the recreation of fault effects in simulation.

4 Practical aspects of FIJI

In this chapter the most important implementation details of the framework are given. First, the messages exchanged between the host and the FI logic are explained. The two following parts describe the inner workings of FIJI's hardware, as well as the software handling configuration, instrumentation, and run-time execution of the FI. Eventually, the small use case intended to allow new users to become familiar with FIJI and experiment with its various options is introduced.

4.1 Communication protocol between host and FIC

Figure 6 depicts the fields in a fault configuration message sent from the host to the FIC. It contains six bits per FIU for the configuration of two fault patterns. A number of padding bits is prepended to the message to align the FIC configuration data (timer values, design ID, ...) to byte boundaries. The exact format depends on the parametrization of the FI hardware subsystem, i.e., the timer width and the number of FIUs.

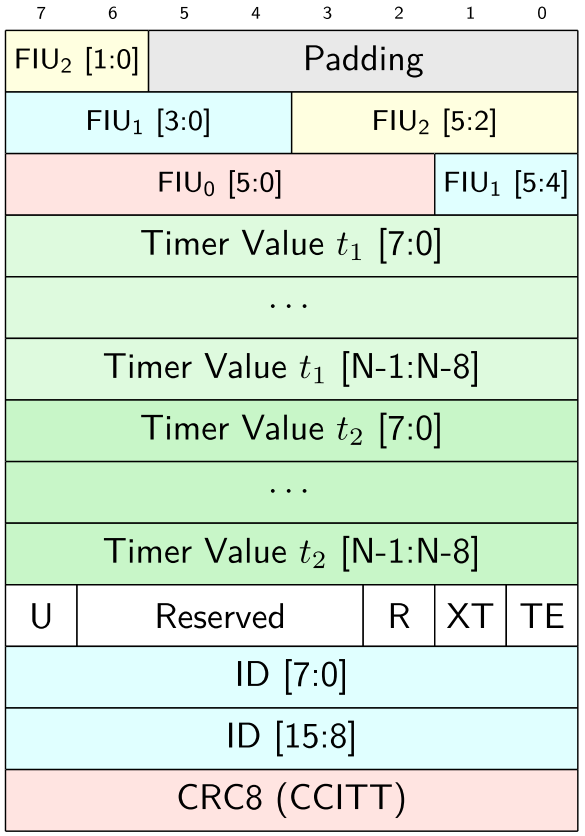

Fig. 6 Host-to-FIJI message for a system with 3 FIUs and *N*-bit timers

The following bytes specify attributes applying to all the instrumented nets for the sequence of phases determined by this instance of configuration message. This data consists of two timer reload values determining the length of the *COUNT* and *FAULT1* phases and a configuration byte. The latter determines the reset (*R*: "reset enable") and trigger behavior (*TE*: "trigger enable", *XT*: "external trigger") of the sequence. Setting bit *U* instructs the FIC to discard the fault patterns and only send a status update back to the host.

The configuration message is concluded with a 16-bit design ID. The purpose of this ID is to prevent the unintentional use of a configuration mismatching the instrumented netlist. For the configuration to be actually applied, this ID must match the one embedded into the design at instrumentation time. The design ID is generated by the Instrumentation Tool by hashing the original DUT netlist and FIJI's parametrization via VHDL constants. Additionally, the integrity of configuration messages is protected via an 8-bit CRC (CCITT polynomial $x^8 + x^2 + x + 1$).

The messages returned by the hardware to the host are single-byte status words as depicted in Fig. 7 that describe the current state of the system. Additionally, the value of two fault detection nets are output that may be used to detect the propagation of an injected error through the design and verify that fault detection within the DUT is working. The integrity of the single-byte status word is protected by a parity bit.

4.2 Fault injection hardware

The HDL wrapper generated by the instrumentation tool instantiates the modified netlist. This netlist has the same input and output ports as the original netlist, but in addition has the broken-up nets exposed as ports, with the original driver now connected to an output, and all the driven cells' inputs connected to an input port.

Legend

P	(even) Parity
Type	`CONF_DONE, READY, UNDERRUN`
F1/2	Fault Detect 1 & 2
U	UART receive error
I	ID mismatch
C	CRC mismatch

Fig. 7 FIJI-to-host message

By parameterizing the wrapper, the entire FI capability can be turned on and off. Either the FI hardware subsystem is instantiated for the instrumented nets to pass through, or each instrumented output is directly connected to the corresponding input effectively reconnecting the split nets (just outside the DUT). This allows the use of the same instrumented netlist of the DUT for FI tests as well as in the final design (i.e., without any FI capabilities).

Additionally, separation constraints can be set to produce a bitstream that contains the DUT and the FIJI logic in distinct physical blocks of the FPGA device and prohibit optimizations beyond DUT boundaries. That way any unintended influences between FIJI and the DUT are avoided. Figure 8 shows an exemplary floorplan with physical separation between the DUT's logic and FIJI.

Details of the FI hardware subsystem can be seen in Fig. 9. In the remainder of this subsection, the functionality and the specifics of the major blocks will be described. The three major components are the UART that establishes the communication to the host PC, the FIUs that

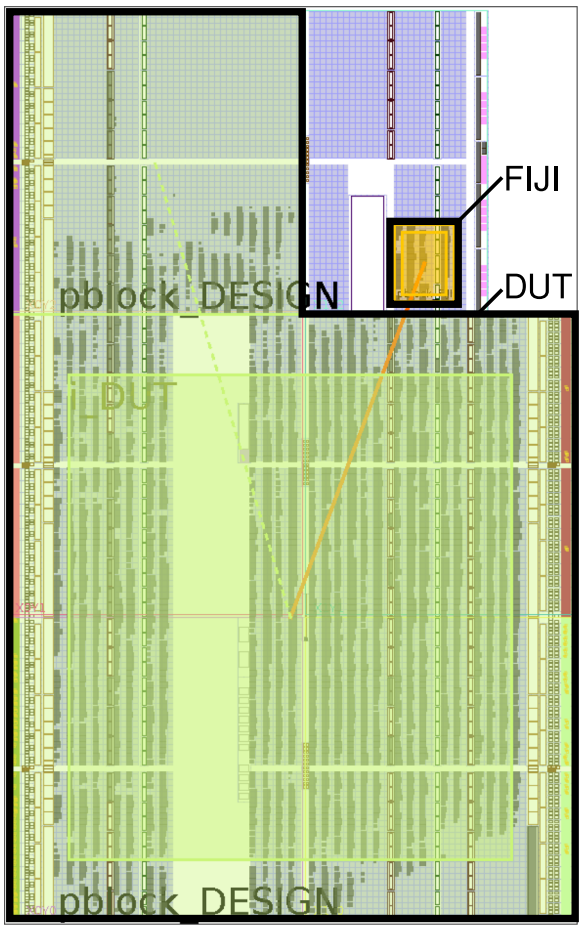

Fig. 8 Example floorplan of an FPGA design containing FIJI

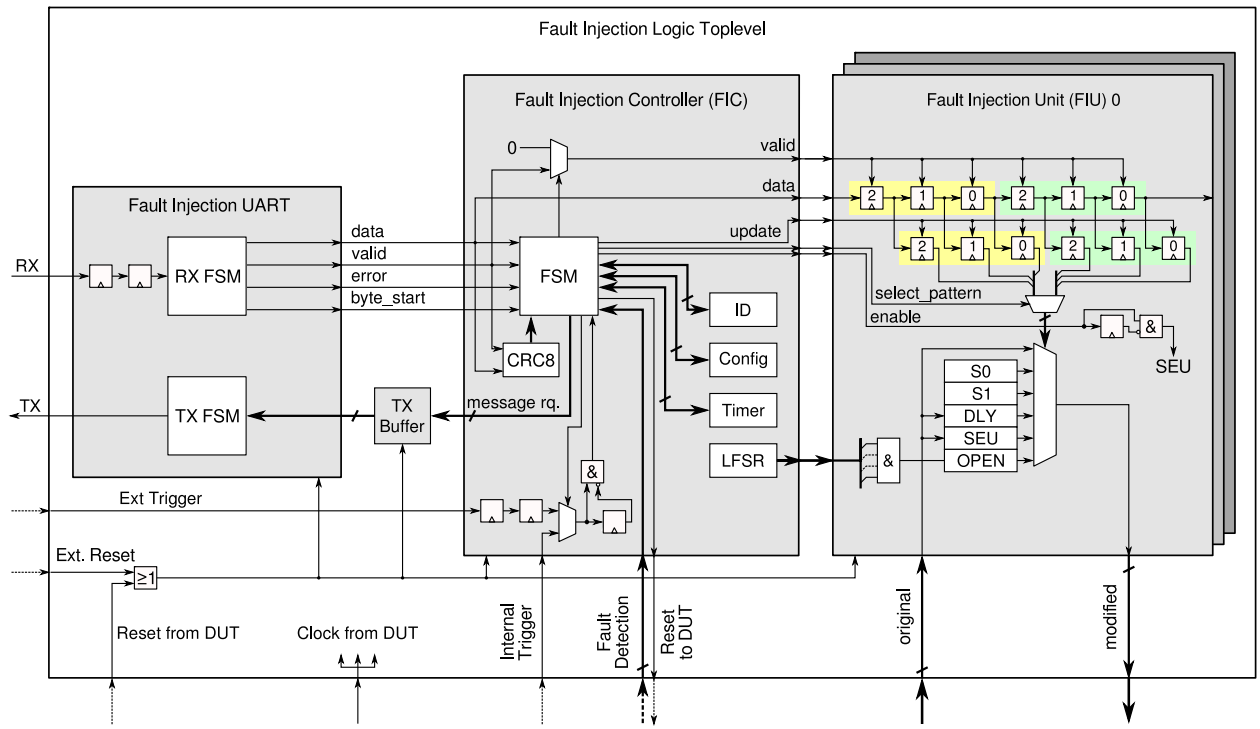

Fig. 9 Overview of the FIJI hardware

directly tap into the instrumented nets, and the FIC that reacts to configuration messages sent by the host and orchestrates the whole FI.

4.2.1 Fault injection controller

At runtime, the behavior of the FIC can be controlled via a serial interface. This interface consists of a data signal, a shift-enable signal indicating that the data line is valid, and a framing signal that indicates the start of a new byte. Additionally, an error signal is used to inform the FIC of detected transmission errors.

A finite state machine (FSM) in the FIC forwards the received data to its internal registers corresponding to the fields in the protocol for the general configuration information and to the FIUs for the fault pattern information. In addition to directing the received data, the FIC also handles the activation of the FIUs, and, if configured, the activation of a reset signal for the DUT according to the trigger and timer values.

In particular, the FIC can be instructed via the configuration message to defer execution of the fault patterns until an edge is registered on either a trigger signal coming from the DUT or an external trigger. The polarity of these edges can be configured at instrumentation time. Timing for the application of the fault patterns is controlled by a timer unit in the FIC. The width of this timer unit can be configured at instrumentation time and is loaded by the FIC with the transmitted timer values at runtime.

Furthermore, the FIC can be configured to generate a reset pulse of adjustable duration if the FIC-to-DUT reset is enabled at instrumentation time. The entire FI hardware subsystem can either be reset by a signal from an external port or from a net of the DUT (the latter, of course, only if the FIC-to-DUT reset capability is disabled).

The internal state of the FIC provides the majority of the status bits transmitted back to the host via the UART (cf. Fig. 7).

Finally, the FIC module also hosts the LFSR that is used by the FIUs to emulate a floating net caused by a stuck-open error. Its width and generator polynomial are selectable at instrumentation time. Each FIU uses a subset of its output bits to create the signal of the “floating” net.

4.2.2 External communication interface

The intended use case for the FI hardware subsystem is to be controlled via a host external to the FPGA. An asynchronous serial interface allows bidirectional communication while requiring just two additional IO pins of the FPGA device and a justifiable amount of FPGA hardware resources.

The UART module handles the synchronization of the incoming data signal as well as the detection of the data framing (start and stop bits). Data is forwarded to the FIC via its serial interface; detected framing errors are signaled using the error line. Furthermore, the parallel outgoing data from the FIC is serialized and a parity bit appended

before it is sent to the host, as a CRC for this direction of data transfer would be inefficient.

During normal operation, a configuration message that is sent by the host to the FIC results in an immediate `CONF_DONE` message by the FIC, followed by a `READY` message at the end of the *fault* phase, i.e., after the timer duration t_1 has been counted down (cf. Section 3.4). Furthermore, an `UNDERRUN` message is sent if the host did not send a new configuration during the first injection phase as specified by t_2.

As both t_1 and t_2 can be as short as a single clock cycle, the `READY` and `UNDERRUN` messages need to be buffered before being sent over the serial link. For this purpose, FIJI instantiates a small FIFO buffer that can hold three messages to be transmitted.

4.2.3 Fault injection units

For each instrumented net, the generated FI wrapper instantiates an FIU. The purpose of each FIU is to inject a fault into its corresponding net when instructed by the FIC. Each FIU is configurable to support at runtime either all fault models (stuck-at-0/1, delay, SEU, stuck-open) or only a single fault model. This reduces the amount of hardware resources and the path delay introduced by the bigger multiplexer thus helping to maintain clock frequencies when instrumenting critical nets.

Each FIU contains two sets of six-bit registers: a shift register that is used to shift in the configuration issued by the FIC, and a pattern register holding the currently active fault patterns. The shift registers in all the FIUs are daisy-chained and filled by the FIC as it receives configuration data (cf. Fig. 6). A separate signal activated by the FIC instructs the FIUs to update the pattern register with the newly shifted-in data. This allows the current configuration to stay active while shifting in new data, as well as discarding an invalid configuration (e.g., if the message's CRC is incorrect) before it affects the DUT.

A multiplexer that is controlled by the currently applied fault pattern bits selects which signal is fed into the DUT. This can be the value of the original net or any faulty version of the signal if a fault is to be injected. The FIU realizes each of the supported fault models as configured. To that end, the multiplexer can select between various flawed inputs, such as constant lows and highs (for stuck-at-0/1 errors), a register with the previous value of the original net (to emulate delay faults), and parts of the LFSR. A user-defined mask is used to customize the frequency and sequence for the simulated stuck-open outputs of the FIUs. This mask specifies which of LFSR bits to AND together to form the faulty signal.

4.3 FIJI software components

The three main software parts of FIJI are written in Perl 5. For graphical user interfaces, they rely on Perl/Tk while netlists are parsed and manipulated with the help of Verilog-Perl [27], which is a library written in Perl and C++ for parsing Verilog and SystemVerilog. Because Verilog-Perl could not distinguish nor manipulate individual signals of concatenations or vectored nets, we added support for these features and contributed it back to the community in an iterating process with helpful reviews from the upstream maintainer Wilson Snyder. The first upstream release including these changes is version 3.440.

The four main Perl programs utilized in FIJI's flow are depicted as reddish blocks with round corners in Fig. 3 and will be described in the following sections.

4.3.1 Setup tool

FIJI Setup (`fiji_setup.pl`, see Fig. 3) is a graphical configuration tool that provides the user a comfortable way to specify the various options of a FIJI project and to save/restore it in/from a text-based configuration file. Figure 10 presents a typical instance of FIJI Setup that depicts the tab used to configure the individual FIUs. On the top and bottom, there are some widgets commonly shown for all tabs. Among other things, the status bar on the bottom shows the deviation of FPGA resources relative to a simple default configuration of FIJI (*virtual resource factors*) to allow the engineer to estimate the resource usage of changed settings without the need to completely synthesize the whole design. The graphical representation of the system shown on the left of the window is also always visible and dynamically rearranged to reflect the current configuration data. This diagram like many other widgets in FIJI Setup provides more detailed information when hovering the mouse over respective elements by way of tooltips. Additionally, all input widgets validate user inputs and signal possible errors or inconsistencies including the causes.

In the FIU tab, which is active in the screenshot, the engineer can create new FIUs or edit, rearrange, and delete existing ones. The select buttons open dialogs allowing to search for elements of a loaded netlist by simple substrings, typical globbing, or regular expressions. They are used to define the instrumented net of the respective FIU (i.e., the locations where to inject faults) and select the driver thereof in case of ambiguity. The *Model* and *LFSR mask* settings refer to the fault model and selection of random bits as explained at the end of Section 4.2.3.

Additionally, the FIUs can be named individually for easier discernibility in the configuration file(s) and latter steps.

4.3.2 Instrumentation tool

The main task of the instrumentation tool (`fiji_instrument.pl`, see Fig. 3) is to actually perform the modifications of the DUT netlist according to the information in the FIJI Settings. In particular, it breaks up

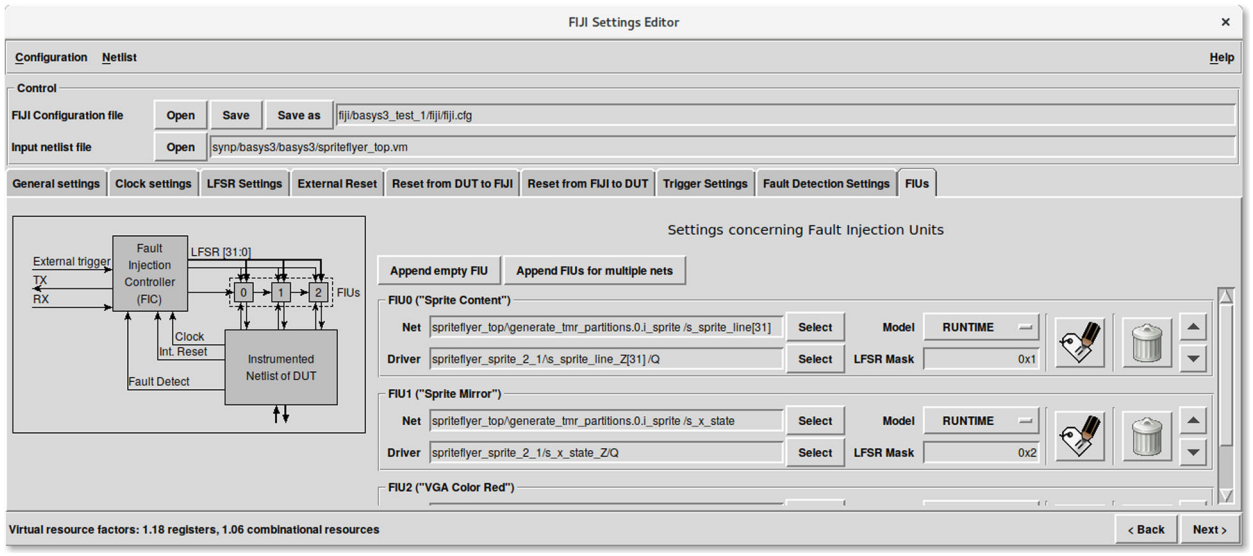

Fig. 10 FIJI setup tool: FIUs tab

each selected net and inserts a *saboteur* in between as follows. User interaction is not required during this process; therefore, the tool was implemented as a command-line program.

For each `[FIUn]` entry in the settings file, it splits the corresponding net into an *original* and a *modified* net as shown in Fig. 11. The driver attached to the *original* net is then connected to a newly created output port of the module where it is instantiated, while the driven pins via the *modified* net are connected to a new input. Both get passed on to the existing top-level entity where they are further exported by adding them to the entities external interface. This *modified netlist* is written to a new file, alongside with a wrapper that instantiates and connects both the *modified netlist* and the parametrized FI hardware subsystem (which includes the FIC and FIUs) as shown in Fig. 2. The wrapper has essentially the same input and output ports as the *original netlist*, allowing to re-use existing pin constraints. It, however, also introduces a small number of additional pins needed by the FI subsystem (e.g., to communicate with the host). Additionally, the instrumentation tool also produces a set of template constraint files for the chosen P&R tool to logically and/or physically separate the FI logic and the DUT netlist.

All steps of the instrumentation need to take busses into account. Moreover, FIJI supports instrumentation of multiple bits of a single bus (with one dedicated FIU per bit). To test our Verilog-Perl changes required to accomplish that as well as the instrumentation code itself, several unit tests were set up consisting of minimal netlists and associated FIJI Settings. These tests cover many different combinations of instrumentation targets (e.g., pins, internal nets) and drivers of these targets which helped to find many bugs in corner cases.

First, the unit tests instrument the netlists with the instrumentation tool. The instrumented netlists are quickly checked for syntax errors and the like by synthesizing them with Synplify and filtering out benign warnings. In addition to the syntax check, a behavioral simulation tries to find discrepancies between an instrumented and untouched entity of the respective netlist. To that end, the test instantiates them in a testbench and compares their output when fed auto-generated input. The tests are not exhaustive but very effective since instrumentation bugs usually affect the signal path in a very direct manner and more formal techniques are unnecessary in this particular use case.

4.3.3 *Execution engine*

While the exact timing of the individual FI phases is handled by the hardware, the FIJI software on the host PC

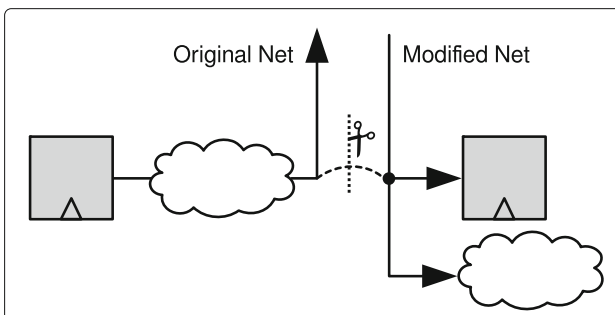

Fig. 11 Splitting of a target net during instrumentation

controls the broader aspects of the execution by providing FI configurations to the hardware (cf. Section 3.4). It is therefore responsible to supply the FIC with new patterns to sustain a cycle-accurate FI if this is desired. This will be discussed in more detail in Section 5.5.

There are two options available to control the *FIJI* logic at run time (as can be seen in Fig. 3). The *FIJI Execution Engine (FIJIEE)* tool is a command-line tool which facilitates downloading pre-defined or random test patterns but can also be controlled interactively. The *FIJIEE GUI* tool (shown in Fig. 12) provides a graphical user interface for roughly the same functionality. Both communicate with the *FIJI* logic in the target FPGA via serial connection (UART) to download test patterns and read back status information.

The *FIJIEE GUI* can create a sequence of fault patterns to be downloaded one after another. For each test pattern, the timer values can be changed, the reset and trigger options enabled, and the individual FIUs configured. Such a sequence of patterns together with settings applying to the whole sequence (e.g., conditions when to halt the test execution) can be saved to a file and reloaded later or exerted by the command-line tool (e.g., in automated tests).

Both applications support a *manual* mode where the parameters of a single fault configuration can be entered and executed. Additionally, *random* tests can be set up where the user determines the global probability of the various fault models as well as lower and upper bounds on t_1 and t_2. The probabilities are then used to determine the state of the FIUs in each message sent in two possible ways: Either a single FIU is selected at random and armed according to the global probabilities, or each FIU is subject to possible faults according to these probabilities, which potentially leads to multiple injected faults in a single phase.

Once a FI test has been completed (either a complete sequence has been downloaded, a fault detect line

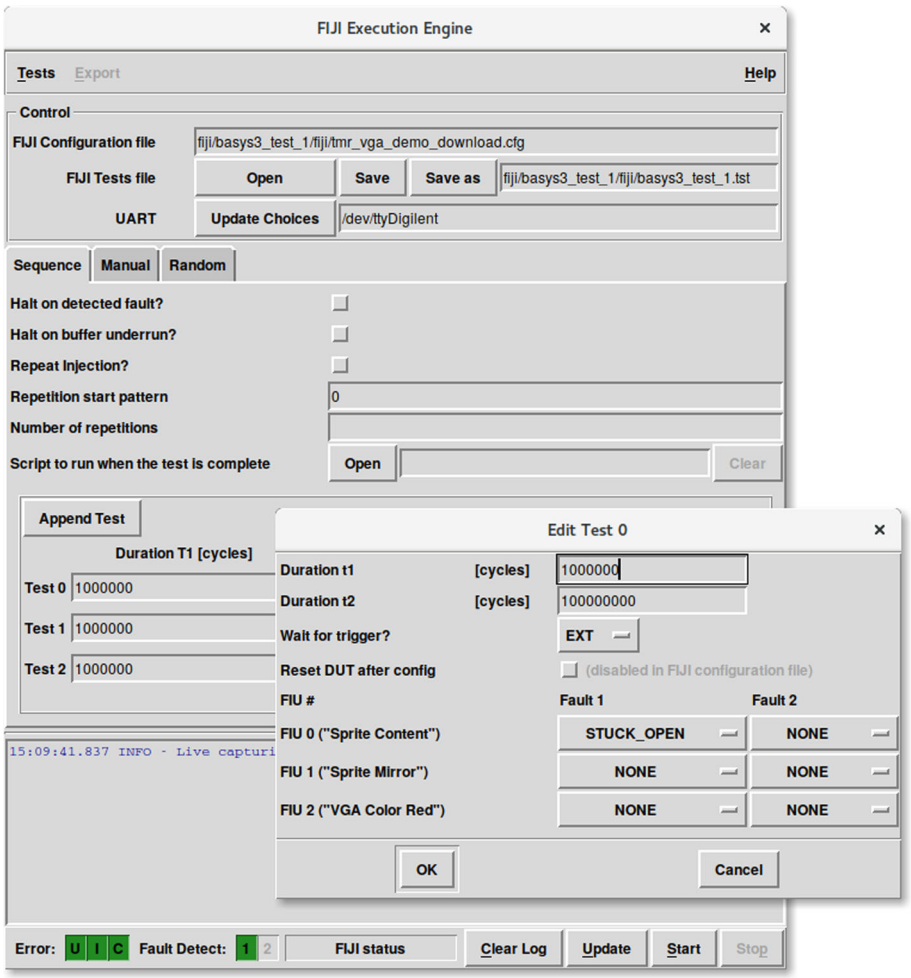

Fig. 12 FIJI execution engine

reported an error, the user aborted the sequence or manual downloading, or a transmission error occurred), the current test run can be exported by the FIJIEE GUI for reproducing it at a later time in hardware or in simulation.

For simulations, two ways are supported: For RTL simulation, FI logic templates can be created that have to be integrated into the various RTL modules manually. Additionally, a scheduling process that controls these templates using hierarchical identifiers is exported.

Alternatively, FIJIEE can also prepare a gate-level simulation for re-executing the test patterns that ran in actual hardware. To that end, *FIJIEE* tools are able to export the executed tests as a VHDL architecture for the top-level entity of the FI logic. This architecture replaces the FIC and the FIUs with a simulation-only description that sets the modified net outputs according to the timing of the test run previously executed in hardware.

4.4 Demo design

FIJI was put to practical use in an FPGA-based safety demonstrator containing a triplicated MC8051 soft CPU core that allows for limited *N*-version programming [28]. As this demonstrator is quite complex and rather heavyweight concerning synthesis tool runtimes, the first public release of FIJI (cf. "Availability of data and materials" section) also contains a reduced version of this demonstrator as a demo design. This design is intended to guide the user through the first steps with the FIJI framework.

The demo design consists of a simple VGA controller that moves a small airplane sprite across the screen. Although initially designed for a few low-cost FPGA boards only (i.e., Terasic DE0 (Altera/Intel Cyclone III), Digilent Basys 3 (Xilinx Artix-7) and Zybo (Xilinx Zynq-7010)), it is easily portable to other development boards that provide a DIP switch, at least two buttons, and a VGA connector.

An overview of the design is shown in Fig. 13. The design contains a module that is responsible for generating the VGA timing signals, as well as row and column information for the *sprite engine* that moves the airplane image across the screen. The output of this sprite engine are the red, green, and blue color signals for the VGA interface. Triple modular redundancy (TMR) has been applied to the sprite engine, with majority voting over each color signal. The voter can be disabled using a DIP switch.

By instrumenting nets in the netlist of this demo design, the user can explore the effects of failures injected into different portions of the design. For example, errors in the VGA timing controller cannot be masked by TMR, making this design unit a single point of failure. Faults in one of the three sprite engine instances can be tolerated without effect if the majority voter is enabled.

Figure 14 shows the behavior of the demo design in simulation under the injection of each supported fault type. The respective faults are injected into the MSB output signal of the *sprite engine*'s shift register that holds the current line of the airplane image. The original value of this signal is denoted as `fiji_s_sprite_line_31_ori_o` in Fig. 14, while the faulty versions of this signal are denoted as as `fiji_s_sprite_line_31_inj_i`. In the demo design (see Fig. 13), this shift register output is used to generate the values of the RGB output registers at each VGA pixel clock cycle. The blue output register is displayed in Fig. 14 as `s_blue_tmr_partitions_1`. In this example, the injected stuck-at and stuck-open faults propagate to the design's output, i.e., they affect the VGA signal. Moreover, the example illustrates that in some cases—if a delay fault or an SEU fault is injected—a fault is masked by the state of the DUT. Thus, the fault effect is visible on the shift register output but does not propagate to the design's boundary.

5 Evaluation

This section provides an overview over empirically determined tool runtimes for a number of designs with different netlist sizes, as well as the influence of FIJI's hardware instrumentation on a design's performance and resource usage. The resource overhead incurred by FIJI is compared to the overhead caused by CrashTest [21] for an OpenSparc T1 use case. Additionally, latency benchmarks in a typical environment (USB-to-serial converter) are provided to allow an estimation of the maximum FI rate.

If not stated otherwise all measurements below were taken on an Intel i7-6600U (2.60 GHz) machine with 20 GiB RAM running Debian GNU/Linux.

5.1 Instrumentation performance

The framework presented in this work is expected to work with netlists that may include tens of thousands of cells, hundreds of modules, and millions of nets interconnecting them. In order to allow estimation of runtimes and memory requirements for real-world netlists, benchmarks were conducted on a set of netlists of different sizes. The netlists were generated by Synplify Pro 2014.09-SP2 from designs included in the Titan Benchmark Suite [29], the AO486 project [30], and the Sparc64 project [31]. In each netlist, FIJI was set up to instrument a single net in the first run. In the second run, ten nets were instrumented. The time and memory (maximum resident set) needed by the instrumentation tool to generate the altered netlist was measured using GNU `time`. The results are shown in Table 1. It can be seen that both the loading time and the time to instrument a given net depend on the netlist size. The netlist size is given in in library cells in Table 1. The netlist's structure also seems to be a factor

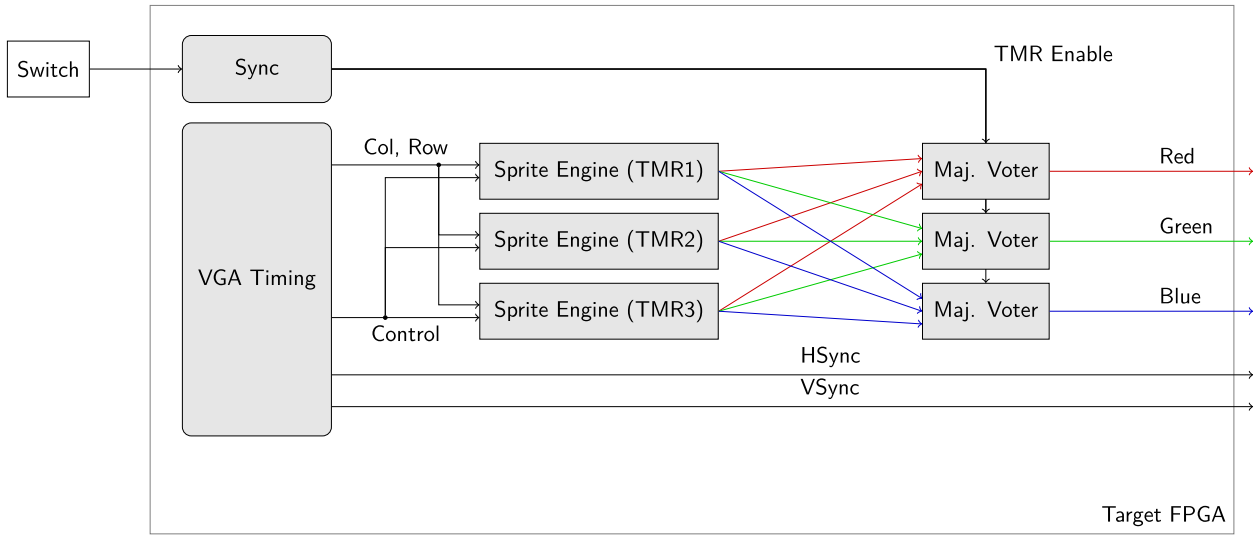

Fig. 13 TMR VGA demo design

in instrumentation time, as the increase between 1 and 10 nets in the larger *ao486_hier* netlist (hierarchical) is by far less dramatic than its flat counterpart *ao486_flat*. This effect is also—albeit not as pronounced—observed in the *sparc64soc_flat* and *sparc64soc_hier* benchmarks. The *bitcoin_miner_hier* benchmark did not complete within the specified timeout of 20 min.

5.2 FPGA resource usage

A minimal system comprising the FIJI control logic and a single FIU requires approximately 300 registers and 300 LUTs in a Xilinx Artix-7 FPGA device. To measure the determining factor influencing the resource usage, we obtained values while varying the timer width (8, 16, 32, and 64 bits), baud rates (115,200 Bd and 3 MBd), and instrumented nets (i.e., the number of FIUs). The clock constraint was set to 200 MHz and the LFSR width was fixed at 16 bit. Figure 15 plots the number of FFs and LUTs over the number of FIUs. The error bars indicate the influence of the other variables, which is comparably small. The measured values clearly indicate the strongest correlation exists between the number of FIUs and the required hardware resources. This is expected since the other variables mentioned above influence only parts that

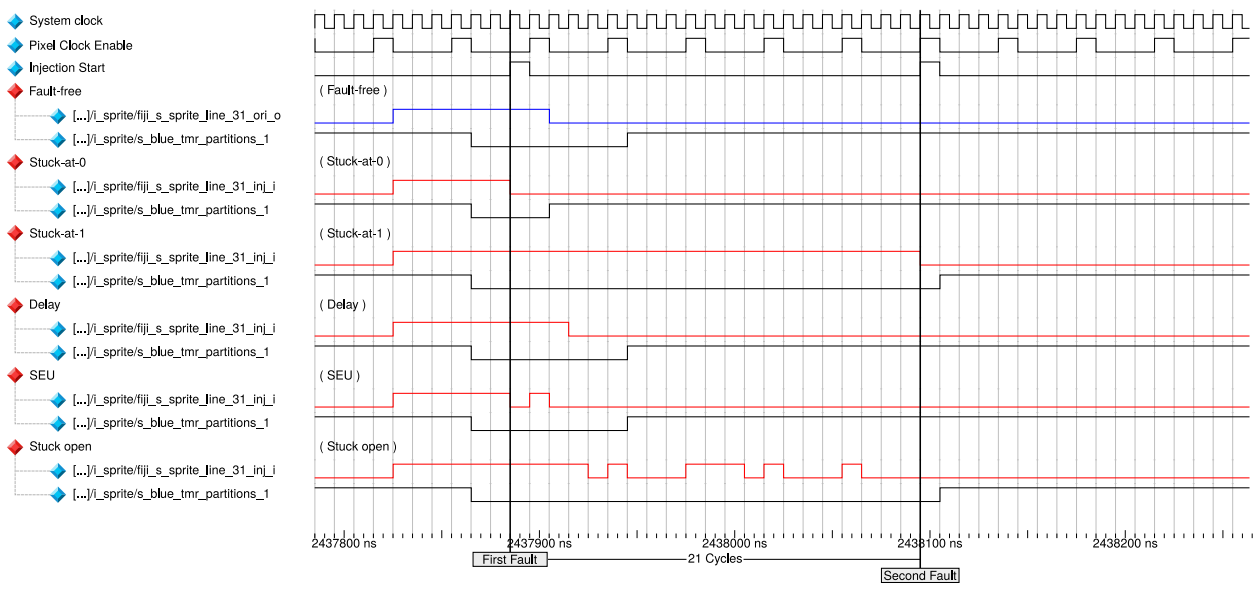

Fig. 14 Behavior of the TMR VGA demo design under fault injection

Table 1 Time and resource consumption of the instrumentation process

Netlist	Library Cells	Time (1 Net) [mm:ss]	Time (10 Nets) [mm:ss]	Peak Memory (1 Net) [GiB]	Peak Memory (10 Nets) [GiB]
leon3_hier	12020	0.11	0.19	0.40	0.41
wb_conmax_top_hier	13118	0.13	0.17	0.50	0.51
ucsb_152_tap_fir_hier	15566	0.14	0.16	0.60	0.61
sudoku_check_hier	19480	0.17	0.23	0.58	0.59
uoft_raytracer_hier	22337	0.19	0.31	0.70	0.71
jpeg_hier	35522	0.29	1.14	0.98	0.99
ch_dfsin_hier	36316	0.32	1.34	1.02	1.03
smithwaterman_hier	49078	0.47	2.54	1.46	1.47
ao486_flat	86309	1.25	5.26	2.52	2.52
ao486_hier	86309	1.04	1.21	2.45	2.46
system90_hier	107021	1.36	1.51	3.77	3.78
sim32_10x10_hier	141491	2.22	2.52	5.36	5.37
LU230PEEng_hier	159822	2.48	7.24	4.94	4.97
sparc64soc_flat	228581	2.42	4.02	5.84	5.83
sparc64soc_hier	228581	3.12	4.33	7.27	7.27
bitcoin_miner_hier	1032434	–	–	–	–

are instantiated exactly once per design while each added FIU directly and indirectly requires a constant amount of additional resources. Other factors that impact the resource count are the configured timer width and LFSR length.

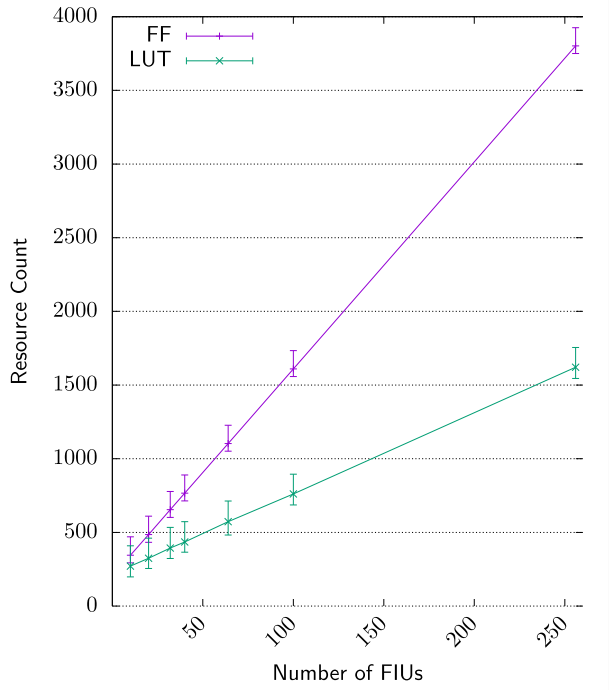

Fig. 15 FIJI resource usage vs. number of FIUs (Xilinx Artix-7 FPGA device)

5.3 Timing penalty

FIJI may impact the timing in two ways: The maximum clock frequency of the FIC and the FIU configuration path, and the logic that is inserted into the path of the instrumented signals. The maximum achievable clock frequency of the FIC itself mainly depends on the width of the timer as shown in Fig. 16. For each timer width, timing results for a Xilinx Artix-7 FPGA device were obtained with designs containing between 10 and 256 FIUs, and at baud rates 115,200 Bd and 3 MBd. The error bars show how the other variables manifest themselves in the timing. Their influence on the timing is greater than on resources because they change the width of some timers too (e.g., slower UART baud rates require a larger counter in the UART).

The logic function inserted into the paths of the original netlist is either an 8-to-1 demux if the FIU's fault model can be set at run-time, or a 2-to-1 demux if the fault model is configured statically. Thus, the timing penalty incurred on each net is constant and does not vary with any other configuration parameter of the design.

5.4 Comparison with CrashTest

Of the netlist-based FI approaches in the current state of the art, *CrashTest* [21] is most closely related to the FIJI framework presented in this work. The main difference to FIJI are the supported fault models and the FI control logic. *CrashTest* uses an embedded CPU (hard-core PowerPC or soft-core MicroBlaze) to control the FI execution, while FIJI provides custom-built logic for this task. Moreover, FIJI allows to instrument individual nets

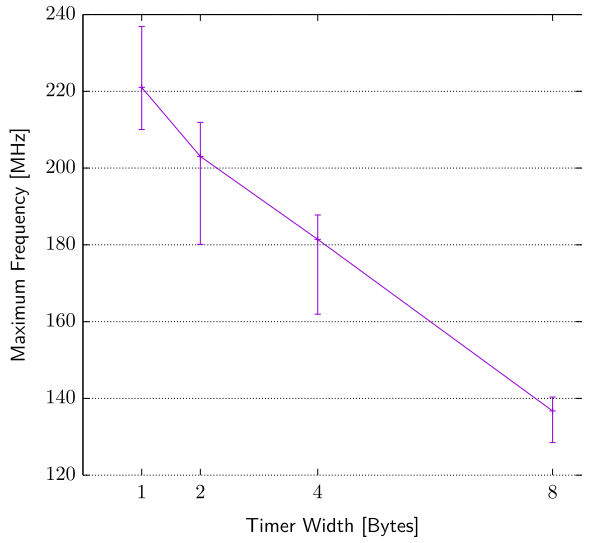

Fig. 16 Maximum frequency vs. timer width in bytes (Xilinx Artix-7 FPGA device)

for multiple fault models that can be selected at runtime, while *CrashTest* only allows to select a single fault model at instrumentation time.

To evaluate FIJI against this pre-existing tool, resource usage and timing were selected as metrics. The authors of *CrashTest* provide use case designs based on an OpenSparc T1 and a Leon3 CPU core on their website [32]. The comparison for this work was done based on the instrumented netlist `sparc_exu_alu.stuck_at_128.ct.v` that has 128 nets in the ALU of the OpenSparc T1 CPU instrumented by *CrashTest*. The same nets were instrumented by FIJI. Both instrumented netlists were synthesized together with the surrounding OpenSparc T1 design, and with FIJI's FI logic in the FIJI variant, in a second step using Synplify Pro. Both designs were placed and routed using Xilinx ISE 11.5, for which the authors of *CrashTest* provide a project that also instantiates the necessary MicroBlaze CPU, memories, and interface logic to the instrumented netlist. For the FIJI variant, these design components were removed, as the wrapper generated by FIJI also contains all necessary FI logic. A resource and timing comparison of *CrashTest* and FIJI is provided in Table 2. The resource values have been reported by Xilinx XPS/ISE 11.5 for a complete system (including all FI control logic). It has to be noted that the MicroBlaze CPU in the CrashTest design may also be used for communication with the OpenSparc CPU, and thus can also be partly seen as a part of the design logic itself. The provided $f_{\max}$ value is the maximum frequency of the instrumented DUT (Sparc) netlist only as estimated by

Table 2 CrashTest vs. FIJI: resource usage and timing

Framework	LUT	Reg	BRAM	$f_{\max}$ [MHz]
CrashTest	40540	32027	111	50.7
FIJI	30567	22285	92	52.6

Synplify Pro 2014.09-SP2, as with *CrashTest*, a clean distinction between fault control logic and the instrumented DUT is not possible after place&route.

In the preliminary work of the authors [24], a comparison of FIJI with other related work has been conducted which also shows the low number of resources consumed by FIJI. For example, Pellegrini et al. [21] make use of an FPGA-internal SDRAM controller that implies a significant overhead of FPGA resources (about 300 LUTs plus 400 FFs at minimum). The approach from Mansour and Velazco [20] even requires a number of external components like a second FPGA, memories, or an Ethernet controller. The same applies to the work of Mogollon et al. [11]. The resource count on LUTs and FFs of FIJI and the work described by Grinschgl et al. [17] is similar to FIJI but; however, a hard-core CPU plus additional Block RAM (to implement the CPU's memories) is needed by [17]. For the bitstream-related approaches [9, 13, 14], a dedicated Altera/Xilinx IP core is required for fault injection that consumes a relatively high number of resources, especially when only a few nets need to be instrumented (5000 LUTs/5400 FFs for Altera's Fault Injection Debugger IP Core, resp. 750 LUTs/350 FFs/7 kByte Block RAM for Xilinx's Soft Error Mitigation IP Core) see [24] for further details.

5.5 Execution latency

A cycle-accurate seamless succession of injected faults as needed when reconstructing a fault in simulation can only be achieved by FIJI when a subsequent fault pattern is ready to be applied upon completion of duration t_2 of the previous pattern (see descriptions in Section 3.4). Thus, t_2 has to be large enough to accommodate the FIC-to-Host latency for the `READY` message, processing time of the host, and the Host-to-FIC latency and transmission time of the next configuration. Figure 17 shows the measured minimal duration t_2 needed to download seamless fault configurations for a given number of FIUs and baud rate reliably (no underruns for 1024 configurations). To reduce jitter in the measurements introduced by the host's operating system the results were obtained while facilitating a kernel with the realtime patch set (Debian Linux Kernel 4.11.0-1-rt-amd64). FIJIEE was executed pinned to a single (exclusively reserved) CPU core with locked memory pages while frequency scaling and scheduling of the process was disabled. As a USB-to-serial converter, the FTDI FT2232C dual-FIFO chip provided

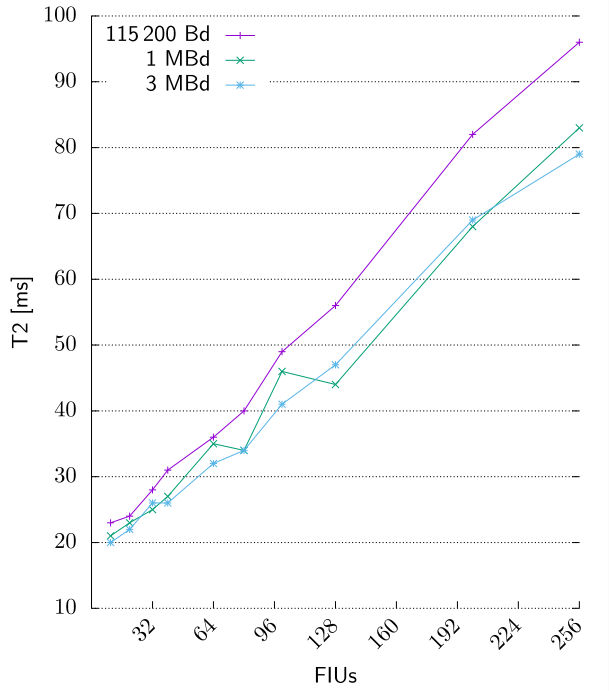

Fig. 17 Minimal duration t_2

by Digilent's Basys 3 development board was used. Minimal t_2 was evaluated for three different common baud rates: 115,200 Bd, 1 MBd, and 3 MBd. It can be seen that even for small configurations (e.g., 10 FIUs), the latency of the entire SW/HW stack limits the minimal t_2 to approximately 20 ms regardless of the baud rate. Starting from there, the minimal t_2 grows linearly with larger configurations due to the additional bits required for each FIU in host-to-FIC messages (cf. Fig. 6). Latency and full-stack data throughput seem to be the limiting factors for minimal t_2 rather than the baud rate at which messages are sent. Due to the modular structure of the FIJI hardware, one approach to increase the number of tests per time unit may be to replace the UART module with another means of communication, e.g., a buffer in on-chip memory that is loaded with configurations for one entire FI campaign. Such approaches would, however, consume a higher amount of FPGA resources.

6 Conclusion and outlook

This paper gives an overview of state-of-the-art methods to inject faults into HDL-based hardware designs. Frameworks manipulating individual signals in HDL code or netlists are described in detail and their pros and cons are discussed. Due to their disadvantages the *Fault InJection Instrumenter (FIJI)* framework was conceived and is made available as open-source software [25, 26]. FIJI is a netlist-based fault injection framework capable of precisely targeting individual signals in a clock-accurate manner with fault models that are switchable during test execution. Its hardware components can be configured depending on the available FPGA resources and timing slack to be tuned to the respective use case. Furthermore, separation constraints can be set to to avoid any unintended influences between FIJI and the DUT that could compromise the certification process of the final system. Additionally, the FI logic can be replaced with simple stubs that forward the unmodified nets. These points make it possible to use almost the exact same design during development, certification and deployment without changing the target FPGA.

FIJI contains extensive documentation including a demo design and automates most of the necessary steps from an existing design to instrumentation of the netlist and execution of fault injection runs. An evaluation highlights its applicability and its ability to inject faults into relatively large designs where gate-level simulations would be infeasible in reasonable amounts of time.

Finally, the following items relate potential for future improvements:

- The instrumentation benchmark results show that in some cases the addition of instrumented nets significantly increases instrumentation time. The overhead of instrumenting each net may be reduced by thoroughly profiling the application in order to speed up instrumentation of a larger number of nets.
- Multi-clock designs where injections into nets of different clock domains are required are not directly supported by the framework. Instrumenting them with FIJI currently requires multiple instrumentation runs to add n fault injection controllers to accommodate faults in n clock domains. At present, FIJI's run-time execution is not able to synchronize fault events between different clock domains, thus losing its clock-accurate injection capabilities.
- In some use cases it would be useful to communicate with the fault injection controller from within the DUT, e.g., by connecting it with selected ports of a hard-core processor within the FPGA for self-diagnosis.
- The available fault models work on the low-level states of individual nets. However, bus systems and other entities driven by FSMs could be more easily tested if they could be dealt with using more abstract semantics. For example, certain protocol errors with multiple conditions on multiple signals could then be injected more effectively.

Acknowledgments
We thank Wilson Snyder for his engagement in shaping our contributions to Verilog-Perl, and our colleagues at the University of Applied Sciences Technikum Wien for their feedback that helped to improve the manuscript.

Funding
This work has been supported by Siemens Austria AG, the Austrian Federal Ministry for Digital and Economic Affairs (BM:DW) and the National Foundation for Research, Technology and Development as related to the Josef Ressel Center "Verification of Embedded Computing Systems" (VECS) and the Josef Ressel Center "Innovative Platforms for Electronic-Based Systems" (INES), managed by the Christian Doppler Research Association.

Availability of data and materials
The initial release of Fault InJection Instrumenter (FIJI) described in this work has been archived [25]. Its parts are licensed under various open-source licenses (main scripts: GPLv1+ or Artistic, documentation: CC-BY-SA, HDL: Solderpad 2.0). For any updates and additional information please visit the project's website [26].

Authors' contributions
All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]University of Applied Sciences Technikum Wien, Höchstädtplatz 6, 1200 Vienna, Austria. [2]Siemens AG Österreich, Corporate Technology, Research Group Electronic Design, Siemensstraße 90, 1210 Vienna, Austria.

Received: 23 October 2018 Accepted: 7 February 2019
Published online: 27 February 2019

References

1. Bobda, C. (2007). *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications, 1st edn*. Dordrecht, The Netherlands: Springer Science & Business Media. https://doi.org/10.1007/978-1-4020-6100-4.
2. Bernardeschi, C., Cassano, L., Domenici, A. (2015). SRAM-Based FPGA systems for safety-critical applications: A survey on design standards and proposed methodologies. *Journal of Computer Science and Technology*, *30*(2), 373. https://doi.org/10.1007/s11390-015-1530-5.
3. Standardization, E.C.F.S. (2008). ECSS-Q-ST-60-02C Space Product Assurance — ASIC and FPGA Development. https://escies.org/download/webDocumentFile?id=19656. Accessed 25 Apr 2018.
4. Kornecki, A.J., & Zalewski, J. (2010). Hardware certification for real-time safety-critical systems: State of the art. *Annual Reviews in Control*, *34*(1), 163–174. https://doi.org/10.1016/j.arcontrol.2009.12.003.
5. Siegle, F., Vladimirova, T., Ilstad, J., Emam, O. (2015). Mitigation of radiation effects in SRAM-based FPGAs for space applications. *ACM Computing Surveys (CSUR)*, *47*(2), 37. https://doi.org/10.1145/2671181.
6. Bernardeschi, C., Cassano, L., Domenici, A., Sterpone, L. (2014). IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, *33*(9), 1342–1355. https://doi.org/10.1109/TCAD.2014.2329419.
7. Asadi, G., Miremadi, S.G., Zarandi, H.R., Ejlali, A. (2003). Fault injection into SRAM-based FPGAs for the analysis of SEU effects, In *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT)*. https://doi.org/10.1109/FPT.2003.1275794 (pp. 428–430).
8. Sterpone, L., & Violante, M. (2007). A New Partial Reconfiguration-Based Fault-Injection System to Evaluate SEU Effects in SRAM-Based FPGAs. *IEEE Transactions on Nuclear Science*, *54*(4), 965–970. https://doi.org/10.1109/TNS.2007.904080.
9. Legat, U., Biasizzo, A., Novak, F. (2010). Automated SEU fault emulation using partial FPGA reconfiguration, In *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. https://doi.org/10.1109/DDECS.2010.5491825 (pp. 24–27).
10. Straka, M., Kastil, J., Kotasek, Z. (2011). SEU Simulation Framework for Xilinx FPGA: First Step towards Testing Fault Tolerant Systems, In *2011 14th Euromicro Conference on Digital System Design*. https://doi.org/10.1109/DSD.2011.32 (pp. 223–230).
11. Mogollon, J.M., Guzmán-Miranda, H., Nápoles, J., Barrientos, J., Aguirre, M.A. (2011). FTUNSHADES2: A novel platform for early evaluation of robustness against SEE, In *2011 12th European Conference on Radiation and Its Effects on Components and Systems*. https://doi.org/10.1109/RADECS.2011.6131392 (pp. 169–174).
12. Azkarate-askasua, M., Iturbe, X., Martinez, I., Obermaisser, R. (2011). *FI4SoC: A fault injection framework for transient fault effects in embedded MPSoCs*, (pp. 81–86). Regensburg: 2011 Proceedings of the Ninth International Workshop on Intelligent Solutions in Embedded Systems. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6086024&isnumber=6086005.
13. Xilinx Corporation (2017). Soft Error Mitigation Controller V4.1. Xilinx Corporation https://www.xilinx.com/support/documentation/ip_documentation/sem/v4_1/pg036_sem.pdf. Accessed 25 Apr 2018.
14. Altera Corporation (2014). Debugging Single Event Upsets Using the Fault Injection Debugger. Altera Corporation. https://www.altera.com.cn/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts_qii53026.pdf. Accessed 25 Apr 2018.
15. Meisner, S., & Platzner, M. (2016). Thread shadowing: On the effectiveness of error detection at the hardware thread level, In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. https://doi.org/10.1109/ReConFig.2016.7857193 (pp. 1–8).
16. Baraza, J.C., Gracia, J., Blanc, S., Gil, D., Gil, P.J. (2008). Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *16*(6), 693–706. https://doi.org/10.1109/TVLSI.2008.2000254.
17. Grinschgl, J., Krieg, A., Steger, C., Weiss, R., Bock, H., Haid, J. (2011). Automatic saboteur placement for emulation-based multi-bit fault injection, In *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. https://doi.org/10.1109/ReCoSoC.2011.5981521 (pp. 1–8).
18. Jeitler, M., Delvai, M., Reichor, S. (2009). FuSE - a hardware accelerated HDL fault injection tool, In *2009 5th Southern Conference on Programmable Logic (SPL)*. https://doi.org/10.1109/SPL.2009.4914906 (pp. 89–94).
19. Zheng, H., Fan, L., Yue, S. (2008). FITVS: A FPGA-Based Emulation Tool For High-Efficiency Hardness Evaluation, In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*. https://doi.org/10.1109/ISPA.2008.46 (pp. 525–531).
20. Mansour, W., & Velazco, R. (2013). An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs. *IEEE Transactions on Nuclear Science*, *60*(4), 2728–2733. https://doi.org/10.1109/TNS.2013.2267097.
21. Pellegrini, A., Constantinides, K., Zhang, D., Sudhakar, S., Bertacco, V., Austin, T. (2008). CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework, In *2008 IEEE International Conference on Computer Design*. https://doi.org/10.1109/ICCD.2008.4751886 (pp. 363–370).
22. Solinas, M., Coelho, A., Fraire, J.A., Zergainoh, N.E., Ferreyra, P.A., Velazco, R. (2017). Preliminary results of NETFI-2: An automatic method for fault injection on HDL-based designs, In *2017 18th IEEE Latin American Test Symposium (LATS)*. https://doi.org/10.1109/LATW.2017.7906748 (pp. 1–4).
23. Pellegrini, A., Bertacco, V., Austin, T. (2010). CrashTest: Fast and Accurate Fault Analysis Platform. https://vhosts.eecs.umich.edu/crashtest/files/docs/CrashTest_v4.pdf. Accessed 25 Apr 2018.
24. Fibich, C., Rössler, P., Tauner, S., Taucher, H., Matschnig, M. (2015). A netlist-level fault-injection tool for FPGAs. *e & i Elektrotechnik und Informationstechnik*, *132*(6), 274–281. https://doi.org/10.1007/s00502-015-0315-4.
25. Tauner, S., & Fibich, C. (2018). FIJI: Fault InJection Instrumenter. *Zenodo*. https://doi.org/10.5281/zenodo.1246734.
26. Fibich, C., & Tauner, S. FIJI (Fault InJection Instrumenter) Website. https://embsys.technikum-wien.at/projects/vecs/fiji.
27. Snyder, W. Introduction to Verilog-Perl. https://www.veripool.org/wiki/verilog-perl. Accessed 25 Apr 2018.
28. Fibich, C., Rössler, P., Tauner, S., Matschnig, M., Taucher, H. (2017). A fpga-based demonstrator for safety-critical applications, In *2017 Austrochip Workshop on Microelectronics (Austrochip)*. https://doi.org/10.1109/Austrochip.2017.13 (pp. 35–40).
29. Murray, K.E., Whitty, S., Liu, S., Luu, J., Betz, V. (2013). Titan: Enabling large and complex benchmarks in academic CAD, In *2013 23rd International Conference on Field Programmable Logic and Applications*. https://doi.org/10.1109/FPL.2013.6645503 (pp. 1–8).
30. `ao486` — an x86 compatible Verilog core. https://github.com/alfikpl/ao486. Accessed 25 Apr 2018.
31. `sparc64soc` — OpenSPARC-based SoC. http://opencores.org/project,sparc64soc. Accessed 25 Apr 2018.
32. CrashTest — A Fast High-Fidelity FPGA-Based Resiliency Analysis Framework. https://vhosts.eecs.umich.edu/crashtest/. Accessed 25 Apr 2018.