Technische Universität Braunschweig

Chair for Chip Design for Embedded Computing

FPGA Accelerated Post-Synthesis Fault Injection for RISC-V Cores

Eike Trumann, Jasper Homann, and Guillermo Payá-Vayá
Chair for Chip Design for Embedded Computing
Technische Universität Braunschweig, Germany
eike.trumann@tu-braunschweig.de

April 4, 2025

Due to the irradiation in space, developing computer systems for space environments requires designing and testing for fault tolerance. How radiation-related faults affect system operation also heavily depends on the software executed. On the one hand, testing for fault induced errors through irradiation experiments is laborious and expensive, and it is only possible once a physical engineering sample has been manufactured. On the other hand, fault simulation is cheaper and available earlier in the development cycle, but it is slow, making full coverage of the fault space infeasible. Fault emulation using FPGAs can be fast enough to fully cover single-bit errors in all flip-flops at the netlist level. This work proposes an FPGA-based methodology for simulating and evaluating single-bit faults in an ASIC implementation of a RISC-V processor core. The RISC-V core's ASIC netlist is re-synthesized for an FPGA device with additional logic for fault injection during emulation time, reaching full coverage of the single-bit fault space in a feasible amount of time. A case study applying the approach to a pipelined RISC-V core implemented on a 45 nm technology and emulated on a Xilinx Zynq-7000 FPGA shows a $2\times$ overhead in flip-flops and a $1.51\times$ overhead in LUTs compared to emulation of the original netlist without fault instrumentation with no reduction of the clock frequency (66 MHz). A total runtime of under 2 minutes is required for a full coverage fault emulation campaign for an exemplary program with an execution duration over 1000 cycles, demonstrating a speedup of up to 100x compared to simulation-based fault injection approaches, making the integration of a continuous fault testing analysis early in the development process feasible.

1 Introduction

Radiation tolerance is a significant concern when developing computer systems for use in space. Circuits and memories can be characterized by testing them while exposed to radiation. However, the effects of radiation on microprocessors are highly dependent on the executed software. Physical radiation testing of each hardware-software combination is laborious and expensive, and low beam time availability can delay projects. This work proposes an FPGA-based fault-emulation methodology for post-synthesis ASIC designs,

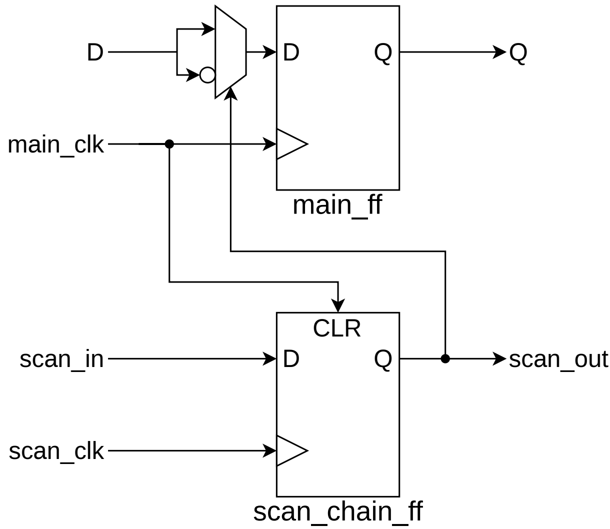

Figure 1: Flip-Flop (main_ff) instrumented with a scan-flop and fault-injection logic.

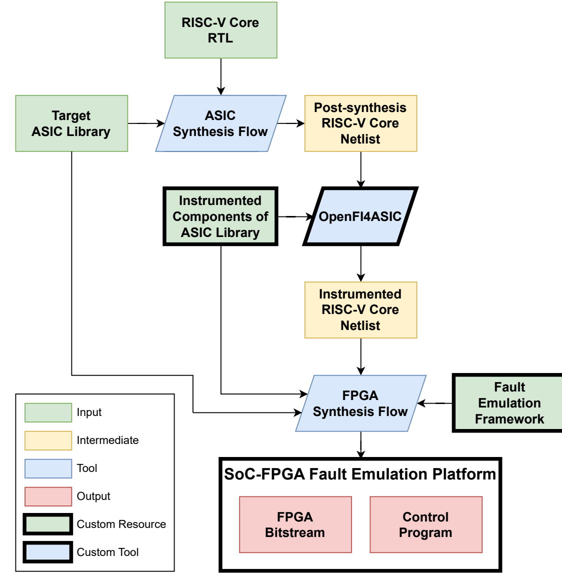

Figure 2: Synthesis and instrumentation flow.

which provides cost-effective and timely full fault injection coverage for single-bit faults in the internal registers and control logic. This allows for continuous software and hardware testing, including single-bit fault injection during development, reducing the risk of unforeseen radiation qualification failure. The methodology is demonstrated in a case study characterizing an open-source RISC-V processor core's single-bit fault tolerance, when executing a simple application. The results provide valuable insights for both application and hardware developers as to where hardware and software fault mitigation techniques can be implemented.

2 State of the Art/Background

Fault injection campaigns using either software simulation or elaborate physical injection methods such as laser fault injection, heavy ion micro-beams, and ASIC chips instrumented for fault injection are established as fault analysis techniques. FPGA-based fault emulation techniques have also been employed. For example, in FT-UNSHADES2 [1], multiple FPGAs and custom circuit boards run fast fault-injection campaigns by inducing single-event upsets through the FPGAs' reconfiguration mechanism. Our previous work FLINT (FauLt INjection Tool) [2] can instrument a design by adding a scan-chain-based runtime fault-injection mechanism. This mechanism enables running fault-injection campaigns with different faults injected at runtime in simulation and on FPGA platforms.

Simulation and emulation techniques can target different chip design tool flow steps. Circuits can easily be simulated and emulated on FPGA devices at the register transfer level. Still, crucial details of the potential resulting hardware are masked, and essential process steps, such as retiming, that could potentially alter the circuit structure are ignored. At the netlist level, the synthesized design can include additional optimizations such as retiming and logic-level radiation hardening techniques. An appropriate simulation library allows netlists to be simulated and resynthesized for FPGA emulation. Although physical hardware simulation at a more detailed level is possible, it is less common for large logic designs due to high computational demands.

3 Implementation

We propose `OpenFI4ASIC` [3] an FPGA-emulated post-synthesis fault-injection framework for any kind of hardware design. However, software developed to be executed on a hardware processor design can also profit of this framework. OpenFI4ASIC is composed of three main components: a fault injection tool, a model for

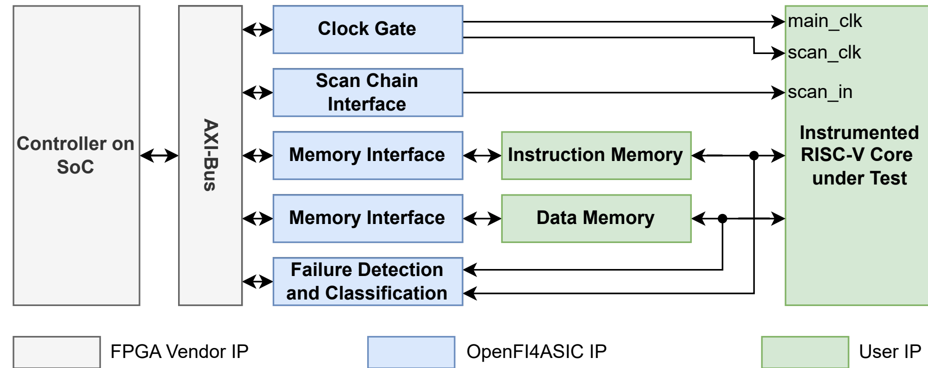

Figure 3: Block diagram of proposed fault-emulation system.

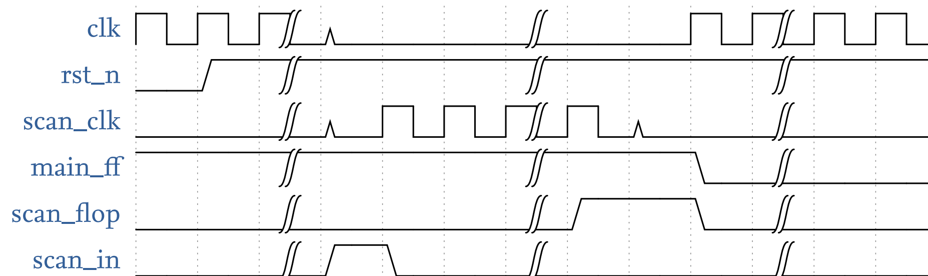

Figure 4: Waveform showing a single fault emulation run.

instrumented standard cells based on our previous work FLINT [2], and an FPGA-based runtime. The fault injection tool-flow and the instrumented standard cell models are generic and can be used on any (digital) hardware design. However, the FPGA-based runtime, presented in this paper, is specifically designed for a Harvard architecture RISC-V processor.

The fault injection tool-flow is used to modify a given netlist, e.g., a RISC-V processor, by replacing selected standard cells with instrumented cells. The instrumented cells are all connected via one or more scan-chains, which control when and how faults are injected. Depending on the model used for the instrumented memory cells (i.e., flip-flops and latches), arbitrary fault characteristics, like bit-flips, stuck-0 and stuck-1, can be emulated.

Figure 1 shows a low-overhead flip-flop model for injecting bit-flips, designed to efficiently use the FPGA resources. Writing 1 to the scan-chain flip-flop, while the main clock is gated, causes the next value to be stored in the main flip-flop to be inverted once the main clock is resumed. Because the scan-chain flip-flop's asynchronous clear is connected directly to the main clock, it is automatically cleared.

Figure 2 shows the tool-flow. The post-synthesis netlist is instrumented via scan-chain insertion using the `OpenFI4ASIC` instrumentation tool, which builds on the original FLINT approach, and the resulting netlist is then re-synthesized with a functional model of the target technology for the FPGA system used for emulation. The instrumented design is then mapped on an FPGA system for fault injection. An integrated hard-core processor of an SoC-FPGA or a soft-core processor mapped on the same FPGA can be used to control the campaign.

Figure 3 shows a block diagram of the fault emulation system. The SoC can control the design and the scan-chain clock, and read and write the instructions and data memory. The FPGA logic manages failure detection and classification by supervising the RISC-V memory accesses. Errors can be detected and classified using a recorded trace or an in-parallel emulated fault-free reference model.

Divergences from the instruction memory trace mean that control flow was potentially compromised.

Table 1: Comparison of resources and frequency between plain and instrumented RISC-V core

	LUTs	LUT Utilization	Flip-Flops	Flip-Flops Utilization	Core Clock Frequency	Scan Clock Frequency
Unmodified RISC-V Core	2873	5.40 %	1706	1.60 %	66.6 MHz	-
Instrumented RISC-V Core	4341	8.16 %	3413	3.21 %	66.6 MHz	400 MHz

Similarly, diverging accesses to the data memory indicate that either wrong input data were read or corrupted data were written. By specifying in which memory locations the program outputs are stored, wrong writes can be further classified by whether they impact critical system functionality, generate wrong data, or write garbage into unused memory regions. While the system provides comprehensive insights into many different fault types, criticality assessment of data errors must be done for each application.

Figure 4 shows how a single fault is injected into a flip-flop in a specific cycle:

1. The system is reset to a known initial state.
2. The main clock is driven until the cycle at which the fault is to be injected.
3. The main clock is paused and a fault injection is initiated by writing a logic '1' into the scan-chain.
4. The scan-chain clock is driven until the fault trigger is stored in the flip-flop instrumented at the targeted flip-flop.
5. The scan-chain clock is stopped, and the main clock resumes for the rest of the system runtime.

This process writes a flipped bit into the main flip-flop, emulating a single-event upset occurring at the start of the cycle. The scan-chain is automatically cleared by connecting the main clock to the scan-flop asynchronous clear pin when the main clock operation is resumed. Arbitrary multi-bit errors can be injected by generating an appropriate pattern for the scan-chain input.

4 Results

As a proof of concept, a 32-bit RV32I 5-stage pipelined RISC-V core was synthesized for the 45 nm NangateOpenCellLibrary using Cadence Genus. The resulting post-synthesis netlist was then instrumented using the `OpenFI4ASIC` fault instrumentation tool.

In order to evaluate the resource overhead, Vivado 2024.1 was used to implement both the unmodified and the instrumented post-synthesis netlist for the `XC7Z020-CLG484` SoC-FPGA. The results are shown in Table 1. The instrumentation overhead is $2\times$ for registers and $1.41\times$ for LUTs. The overhead in the LUTs is due to the additional multiplexer introduced in the instrumented flip-flop and the logic required to emulate the clock-enable and reset behaviour of the original flip-flop. The instrumentation does not impact the maximum clock frequency. The scan-chain can be driven at a significantly higher maximum frequency of 400 MHz.

The average runtime of a single fault injection in the flip-flops is $T_{FI} = L \times \frac{1}{f_{core}} + \frac{1}{2} N_{ff} \times \frac{1}{f_{scan}}$, where L is the program length in cycles, N_{ff} is the number of instrumented flip-flops and f_{core} and f_{scan} are the core and scan-chain clock frequencies, respectively. The average runtime of a single fault injection in the memory is the time required for one read and one write access from the SoC through the AXI-Bus (see Figure 3). Full single-bit error fault space coverage requires the execution of $L \times (N_{ff} + N_{mem})$ fault injections. For our core with $N_{ff} = 1706$, $N_{mem} = 3200$ and an exemplary program with $L = 1000$ and 100 instructions, we estimate a runtime of the full coverage fault emulation campaign of under 2 min.

```c
__attribute__((section(".text.start"), naked))
void _start() {
  asm volatile(
    "la sp, _estack; call _main; call _halt"
  );
}


void _main(void) {
  int volatile *input = (int volatile *)0x00;
  int volatile *result = (int volatile *)0x04;
  *result = fib(*input);
}


__attribute__((noinline))
int fib(int n) {
  volatile int a = 0;
  volatile int b = 1;
  volatile int c = 0;
  for (volatile int i = 0; i < n; i++) {
    c = a + b;
    a = b;
    b = c;
  }

  return c;
}


__attribute__((naked))
void _halt() {
  asm volatile("halt:\n\t j halt");
}
```

Listing 1: Iterative fibonacci C code

```asm
# 00000010 <_main>:
  addi sp, sp, -0x10
  sw   ra, 0xc(sp)
  lw   a0, 0x0(zero)
  jal  0x30 <fib>
  sw   a0, 0x4(zero)
  lw   ra, 0xc(sp)
  addi sp, sp, 0x10
  ret

# 00000030 <fib>:
  addi sp, sp, -0x10
  sw   zero, 0xc(sp)
  li   a1, 0x1
  sw   a1, 0x8(sp)
  sw   zero, 0x4(sp)
  sw   zero, 0x0(sp)
  lw   a1, 0x0(sp)
  bge  a1, a0, 0x84 <fib+0x54>
  lw   a1, 0xc(sp)
  lw   a2, 0x8(sp)
  add  a1, a2, a1
  sw   a1, 0x4(sp)
  lw   a1, 0x8(sp)
  sw   a1, 0xc(sp)
  lw   a1, 0x4(sp)
  sw   a1, 0x8(sp)
  lw   a1, 0x0(sp)
  addi a1, a1, 0x1
  sw   a1, 0x0(sp)
  lw   a1, 0x0(sp)
  blt  a1, a0, 0x50 <fib+0x20>
  lw   a0, 0x4(sp)
  addi sp, sp, 0x10
  ret
```

Listing 2: Compiled code for iterative fibonacci

4.1 Characterisation of a RISC-V Core

To demonstrate the proposed methodology, a fault emulation campaign was run on a custom 5-stage pipelined RV32I M-Mode Harvard architecture RISC-V processor core [4]. Listing 1 shows the simple iterative Fibonacci implementation used as an exemplary test program. The code was compiled for `rv32i` with `clang 18.1.3` with full optimizations (`-O3`). To increase the observability of the program, all variables are declared `volatile`, such that no memory accesses are elided. This is done to specifically test the core's memory interface and forwarding mechanism. Listing 2 shows part of the disassembly of the compiled binary. It can be seen that only the registers `ra`(`x1`), `sp`(`x2`), `a0`(`x10`), `a1`(`x11`) and `a2`(`x12`) are used.

Figure 5 shows the results of the error classification. For each cycle and each flip-flop, a fault injection is performed (i.e., fully covering the single bit-flip fault space). The resulting errors are classified into data

Table 2: Error rates by flip-flop category and for all flip-flops in the processor core.

Category	Data Flow	Control Flow	Both Errors	No Error (Masked)
All	6.21 %	4.09 %	3.84 %	93.54 %
gp_register_file	4.44 %	4.16 %	4.14 %	95.54 %
csr_register_file	0.00 %	0.00 %	0.00 %	100.00 %
pipeline_datapath	7.41 %	2.47 %	2.06 %	92.19 %
pipeline_controlpath	17.48 %	7.24 %	6.62 %	81.90 %
PC	27.91 %	28.29 %	23.47 %	67.27 %

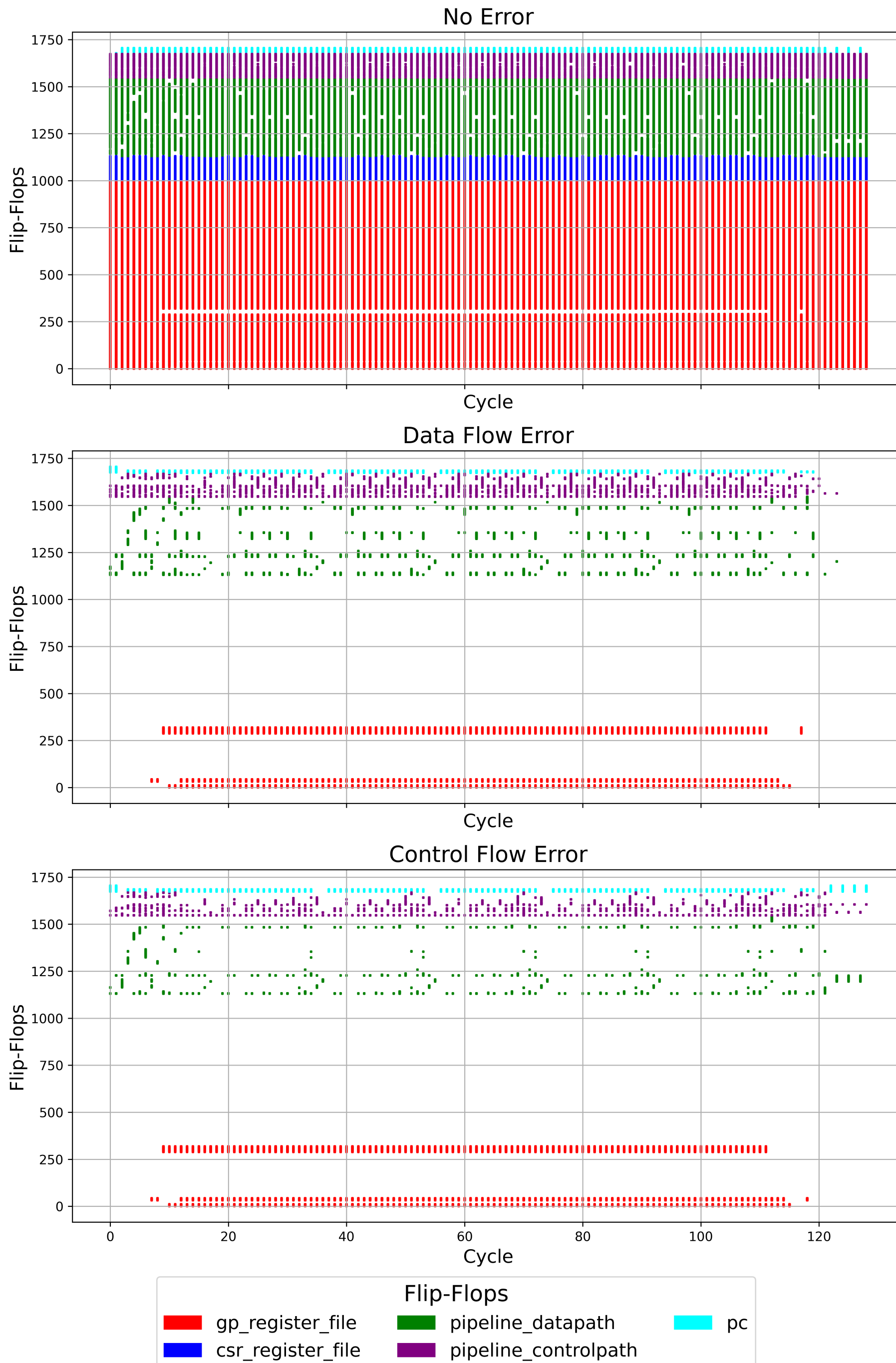

Figure 5: Scatter plot of control and data flow errors, when injection a single bit-flip in a specific flip-flop (y-axis) and a specific cycle (x-axis).

errors (any mismatch in the data memory), control errors (i.e., the final PC differs from the reference), or both. The flip-flops are categorized based on their function as part of the processor microarchitecture. The flip-flops are additionally sorted such that the bits belonging to the same register are grouped together with the least significant bits at the bottom. Table 2 shows the aggregated error rate for each category and the complete core.

From these results, the following observations can be made:

Table 3: Emulation vs. simulation absolute and relative performance

Method	#FFs	#Cycles	#FIs	Runtime	Time/FI	Speedup
Simulation	100	100	10,000	243.985 s	24 ms/FI	1x
Questasim	1706	130	221,780	5411 s (est.)	24 ms/FI	1x
Emulation	100	100	10,000	1.661 s	0.166 ms/FI	146.89x
Zynq 7000	1706	130	221,780	43.418 s	0.196 ms/FI	124.63x

- Flip-Flops storing data not used in the program, i.e., unused general-purpose registers and all control and status registers, do not cause errors when flipped.
- Even though register `a2(x12)` appears to be used by the program, faults injected into its flip-flop do not cause errors. The data are directly forwarded from the load-store unit and thus never read from the register file bits.
- Because only the lower address bits are used to address the memories, faults injected into the unconnected bits of the PC and registers `ra(x1)` and `sp(x2)` used only for storing instruction and memory addresses respectively do not cause any errors.
- Two bits in the control path cause an error when flipped in almost any cycle. These bits are part of the interrupt mechanism and flipping them causes either a trap or a return from a trap, which, in this experimental setup, are equivalent to a jump to address zero.
- What faults cause errors depends on the type of instructions executed. The results shows larger errors in the data and control path during startup and at the end of each loop iteration.

4.2 FPGA Emulation vs. Simulation Performance

To measure the speedup achieved by using fault emulation vs. fault simulation, we simulated a simplified model of the fault emulation system using a VHDL testbench instead of a C program for control and error classification. This model was optimized and simulated using Questasim 2025.1 using a single run for all injected faults on a DELL Latitude 7450 with an Intel Core Ultra 165H at 4.4 GHz turbo and 32 GB main memory. It is worth mentioning that hardware simulation using Questasim only utilizes a single thread. As a benchmark, the EIS-V processor with a total of 1706 flip-flops and the iterative Fibonacci application was used, but faults were only injected in the first 100 flip-flops and cycles. Additionally, we measured the complete runtime of the fault emulation in each flip-flop and cycle with both the processor and scan-chain clocks at a frequency of 50 MHz. For the simulation, the runtime for the full fault injected was extrapolated, presenting a best-case estimate due to faults in flip-flops later in the scan-chain taking longer to inject bit-flips into.

Table 3 shows the absolute and relative performance of both, simulation and emulation, for the benchmark for an entire run. The emulation achieves a speed-up of more than 120x compared to the simulation. This could be further improved by increasing the scan-chain clock frequency to 400 MHz and by implementing a snapshotting mechanism to reduce the overhead of repeated fault-free execution.

5 Conclusions

The proposed fault emulation methodology provides a fast and cost-efficient solution for evaluating changed fault characteristics, both for possible modifications of the program code and the hardware architecture of the processor core. The methodology has been demonstrated on a custom RISC-V processor, running a simple

application, showing that the use of FPGA emulation enables full coverage of the single bit-flip fault space in reasonable runtimes.

The use of `OpenFI4ASIC` facilitates the analysis of reliability statistics for different program implementations and different RISC-V processor cores. Moreover its fast executing time allows its use even in early design stages. Future enhancements in `OpenFI4ASIC` are planned, including support for longer-running programs through a snapshotting mechanism, increasing parallelism, and incorporating fault-masking terms to reduce the fault space.

References

[1] J.M. Mogollon, H. Guzmán-Miranda, J. Nápoles, J. Barrientos, and M.A. Aguirre. 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*, pages 169–174, 2011.

[2] Rochus Nowosielski, Lukas Gerlach, Stephan Bieband, Guillermo Payá-Vayá, and Holger Blume. Flint: Layout-oriented fpga-based methodology for fault tolerant asic design. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 297–300, 2015.

[3] Jasper Homann, Eike Trumann, Lucian Lohse, Moritz Weibrich, Christian Ewert, and Guillermo Payá Vayá. OpenFI4ASIC. https://github.com/tubs-eis/OpenFI4ASIC. Software. License: MIT.

[4] Jasper Homann, Gia Bao Thieu, and Guillermo Payá Vayá. EIS-V. https://github.com/tubs-eis/EIS-V. Software. License: MIT.