Technical University of Munich
Department of Electrical and Computer Engineering
Chair of Electronic Design Automation

Fast RTL-based Fault Injection Framework for RISC-V Cores

Master's Thesis

Johannes Geier

Technical University of Munich
Department of Electrical and Computer Engineering
Chair of Electronic Design Automation

Fast RTL-based Fault Injection Framework for RISC-V Cores

Master's Thesis

Johannes Geier

Advisor :	M.Sc. Uzair Sharif
Advising Professor :	Prof. Dr.-Ing. habil. Daniel Müller-Gritschneder
Topic issued :	01.09.2019
Date of submission :	06.03.2020

Johannes Geier
Hopfenstraße 29
85309 Pörnbach
johannes.geier@tum.de

Abstract

The growing demand of processing power in an increasing number of embedded systems deployed in various circumstances has led to new challenges towards the development of such systems. One of these challenges is a digital system's resiliency against soft errors that can alter internal states and lead to unforeseen and sometimes critical behavior. Simulating such errors at a system's early design phases can help with integrating and evaluating countermeasures. This work introduces a fast Register Transfer Level (RTL)-based fault injection framework for soft-error evaluations of RISC-V processor cores. To enable injections, a tool was developed that transforms the RTL model and builds a specific injection Application Programming Interface (API) from the core's hardware description. The modified RTL and built API were put to test in a proof of concept framework evaluating an open-hardware RISC-V core's behavior towards random bit flips in its internal states. The framework is characterized by its utilization of automatically generated, however, core-specific sources which enable a high fault injection capability while maintaining low simulation overhead. The evaluation results were analyzed with respect to the probability of a certain error occurring after a random bit flip and the probability of a specific injection target being the cause for a certain error.

Contents

List of Figures

List of Tables

Listings

Acronyms

API	**A**pplication **P**rogramming **I**nterface
AST	**A**bstract **S**yntax **T**ree
CPU	**C**entral **Processing** **U**nit
CSR	**C**ontrol and **S**tatus **R**egister
FSM	**F**inite and **S**tate **M**achine
GCC	**G**NU **C**ompiler **C**ollection
GNU	**G**NU **D**ebugger
GPR	**G**eneral **P**urpose **R**egister
HDL	**H**ardware **D**escription **L**anguage
IPS	**I**nstructions **P**er **S**econd
ISA	**I**nstruction **S**et **A**richitecture
ISS	**I**nstruction **S**et **S**imulation
LSU	**L**oad-**S**tore **U**nit
MPU	**M**emory **P**rotection **U**nit
MSB	**M**ost **Significant** **B**it
OS	**O**perating **S**ystem
PULP	**P**arallel **U**ltra **L**ow **P**ower Platform
PRNG	**P**seudo **R**andom **N**umber **G**enerator
RISC	**R**educed **I**nstruction **S**et **C**omputer
RTL	**R**egister **T**ransfer **L**evel
SET	**S**ingle **E**vent **T**ransient
SoC	**S**ystem **on** **C**hip
UI	**U**ser **I**nterface
VP	**V**irtual **P**rototype
VRTL	**V**erilator **R**egister **T**ransfer **L**evel
VSoC	**V**irtual **S**ystem **on** **C**hip
XML	e**X**tensible **M**arkup **L**anguage

1. Introduction

1.1. Motivation

Most physical processes are controlled by information technology systems. Decentralization of control systems, access through large scale networks and more complex application software and inclusion of full-scale operating systems (OS) have lead to generally higher loads on embedded systems. This growing demand of processing power for a previously reasonably complex part of modern electronics has been met by the possibility to feature a much greater amount of components in an increasingly smaller form factor. At the same time, testing and even more so verification struggles to keep up with this growth. One way to encounter this development can be to utilize virtual prototypes (VPs) of such systems in early stages of design.

While designer's of application-level software use VP sockets to study and optimize properties of code and its means of employment, hardware developers might have to look into more than just the behavior of cross-compiled code. Since the deployed unit will manipulate our physical world, the same can be the case vice versa in an unintended way. A case of this are soft errors, where data storage elements inside the digital system are corrupted by unpredictable manipulators. These errors normally do not damage the system's hardware, however, might lead to erroneous behavior, thus, output of the system to the physical world. Soft errors can manifest themselves often by biased or unbiased bit-flips in sequential logic caused by radiation particles hitting such elements. Elaborating the impact of these errors on the system's application is a major concern on the design strategies which questions the coverage of simulating soft errors in virtual prototypes.

Current research tasked itself not only with investigating the impact of soft-errors but also how one should approach their modeling with respect to the level of abstraction. The most prominent idea is to do so on a functional or instruction level. There, instructions of the simulation target instruction set architecture (ISA) are translated just-in-time to equivalents of the host's ISA, where the simulation is performed. Prominent examples for this type of are QEMU and ETISS (TUM EI EDA n.d.). A more detailed approach is taken, when instructions are not only translated to the host ISA but interpreted. Here, the resulting traceable and instruction-wise execution emulates the ISA behavior (SiFive n.d.). Both ideas have in common that the smallest notion of time is one target instruction, therefore, can be summarized as instruction set simulation (ISS). A more fine-grained abstraction level is register transfer level (RTL). While ISS only mimics the system's behavior, an RTL simulation imitates the hardware itself. To clarify the dimensions of those levels, a gearbox simulation could be imagined. Depending on the observational goals, a simple equation describing how speed and momentum are translated from input to output could be enough. For processing systems, such equations would correspond to the ISS level. This equation will not yield any information about the gearbox's inner workings that enable the translation nor how certain parts interact with each other. With the assumption of an unbreakable gearbox, an equation would be enough. However, problems arise when a designer wants to introduce faults, where no information about alterations to the gearbox's behavior is known, e.g. a gear's teeth breaks and there is no information about how this affects the translation. In contrast, simulating the gearbox more precisely would not depend on information about the system's changed behavior as the new behavior is described through the changes in the hardware itself. For processing systems, an executable model of the gearbox's hardware would correspond to RTL.

Since the smallest notion of data storing elements in ISS are general purpose registers (GPRs) and optionally control and status registers (CSRs), a large portion of possible soft error victims are left out from simulation. Research has shown that high-level error injection methodologies, i.e. only injecting in registers and memory, can lead to highly imprecise results (Cho et al. 2013). Therefore, RTL would be the obvious choice for the level of abstraction in soft error simulations. However, a more fine-grained simulation comes with costs. In this case, computational effort to simulate these additional hardware components. Minimizing this simulation effort by modifying highly optimized cycle-accurate RTL simulators for soft error evaluation is the focus of this work.

1.2. Contributions

In this work, the generation of an executable RTL model for a Reduced Instruction Set Computer (RISC) that enables guaranteed fault injection is explored. The focus lays on;

- understanding Verilator as the main tool building the host-executable model from the hardware description language (HDL) sources,
- modifying the Verilator RTL model in order to enable cycle-accurate, and target-explicit injection of errors,
- proposing a proof-of-concept RTL fault injection toolchain that builds an injection framework from HDL sources and with minimal manual input,
- interpreting the results of a fault injection evaluation for a RISC example (RISC-V core).

1.3. Tasks

First, the tool generating the executable model (VRTL), Verilator, has to be explored. This concludes understanding the VRTL to find ways to perform guaranteed fault injections on fault injection viable components, exclusively. Thus, injection targets have to be identified, located and classified in order to optimize the overall simulation process by avoiding injections in model variables that do not correspond to real-world soft error victims.

Next, an algorithm must be developed and implemented that modifies the VRTL sources to allow cycle-accurate injections. Then, the modified VRTL is incorporated in a simulation and fault injection framework for a RISC-V core enabling execution of cross-compiled application code and fault injection scheduling. At last, space (bit) and time (execution cycle) random faults are injected and their impact on the processor core analyzed.

2. Background

2.1. Fault Model

2.1.1. Soft Errors

As a type of transient errors in digital systems, soft errors can be mostly traced back to effects of neutron and alpha particle impacts on circuitry. The resulting fault can be modeled as a single event transient (SET) (Maniatakos et al. 2011) or sometimes also referred to as single event upsets (SEUs) (Baumann 2005). SEUs can affect combinational or sequential logic. However, sequential soft error victims, such as flip-flops and D-latches, have more impact on a system's behavior since their state is stored in-between system clock cycles and their state is only updated during a short latch-up phase. On the other side, combinational logic is continuously driven and converging to a stable state, thus, should have an inherent resiliency against SEUs where the transient impact is much smaller than the clock period. In designs with large propagation delays, thus, low clock frequencies, a SEU seldomly has an impact on combinational logic since the SEU has to happen during the relatively short latch-up phase. However, in high frequency designs a SEU might propagate more easily through the combinational logic, thus, increasing the chance of latching the error (Baumann 2005). Correct investigation of soft errors on combinational logic requires either a gate level model of the system or a RTL model with a modified net list. For this thesis, the following assumptions are made regarding the soft error model:

i) SEUs in combinational logic are disregarded.

ii) SEUs are modeled as bit flips in data storage elements.

iii) Bit flips occur right after a correct latch-up.

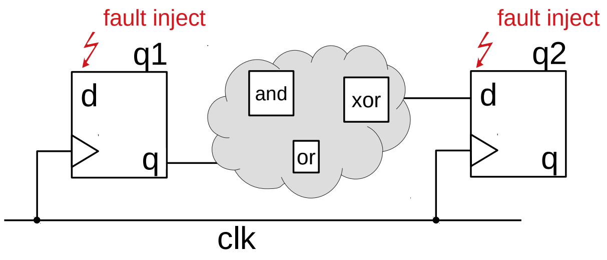


Figure 2.1.: Simple register stage

Fig. 2.1 shows a simple register stage. Register `q2` is driven by combinational logic fed by `q1`. Both `q1` and `q2`, are possible victims of a soft error modeled by a fault inject.

2.1.2. Injection Model

Considering the RTL representation of a synthesized digital systems, thousands of nets and registers can be identified. Exploring a spatial and time explicit SEU, requires defining preliminary modeling rules for fault injections. In this thesis, an faulty model FI for one SEU is described by

$$FI = \{M, b, c, T\}$$

Variable b refers to a single-bit of a clocked data storage element in the bit-representation of a model M, e.g. a micro-architectural register, and c to a specific clock cycle in the observation period. The injection type T describes whether the target's value is flipped unbiased or biased towards a specific state, i.e. a logically high register injected with a high-biased injection, would not be affected, whereas a logically low one would flip to logic high. However, in this thesis, only unbiased flips are considered.

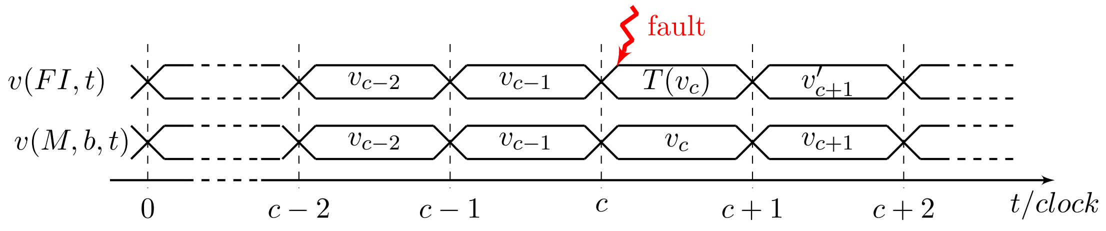


Figure 2.2.: Single-bit target fault

Fig. 2.2 shows a timing diagram for a single-bit fault injection. Equation $v(M, b, t)$ describes the clean value of bit b of model M over a simulation time t. $v(FI, t)$ refers to the value over simulation time with the faulty model. To guarantee an injection, the assumption is made, that at the moment of injection the target is stable, i.e. retaining the faulty state $(T(v_c))$ at least until the next clock cycle.

2.1.3. Fault Effect Classification

SEU do not necessarily produce errors in the affected system. Furthermore, if an error is generated, its observed impact during and after simulation time can vary. Based on the idea of basic fault effect classes described in (Cho et al. 2013), this work handles fault effects through the following after-simulation-time classification:

- **Application Errors:** The application finishes, however, output or program flow is unexpected and varies from an error-free reference run. Additionally, in-system hardware is not capable of detecting these errors.

  (AO) **Output**: The output memory differs from an expected, golden value. This includes peripherals' in a System on Chip (SoC) context.

  (AP) **Program Flow**: The program execution trace varies from an expected one. Considering these errors often is important for applications demanding a specific execution time, e.g. control applications, where the controller is designed with a worst case execution time in mind.

- **System Errors:** The system detects its erroneous behavior. Exceptions can be issued and could then be handled by dedicated processes in software.

  (SM) **Memory Access**: An invalid access to a protected memory region was detected by a dedicated hardware component, e.g. a Memory Protection Unit (MPU).

  (SI) **Illegal Instruction**: An invalid instruction in the pipeline has been caught.

- **Application Finished Correctly:** The program finishes as expected and no errors were detected.

  (MSK) **Soft Error Masked**: After execution no difference to the reference run can be identified. This means, output memory, execution cycles, and micro-architectural state of the model match the reference.

  (NSK) **Soft Error Not Masked**: After execution no difference to the reference run can be identified. However, the micro-architectural states differ from an expected one.

- **Hang-up Error:** The application does not finish and no system error is detected.

  (HL) **Logic Stall:** The pipeline is active, i.e. instructions still progress through the pipeline, however, no application progress can be made. Reason can be that the program is stuck in an endless loop because the loop's exit check continuously fails. Both software, e.g. an OS, and hardware, e.g. a watchdog, might be able to solve this issue.

  (HP) **Pipeline stall:** The pipeline is inactive, i.e. continuously stalled. This state can only be left by reset.

2.2. Verilator RTL

2.2.1. Verilator

The open-source tool Verilator synthesizes hardware descriptions, specified in Verilog and SystemVerilog hardware description language (HDL), to executable C++ or SystemC modules. The more general implementation as a source library enables a convenient integration of Verilator RTL (VRTL) in various applications.

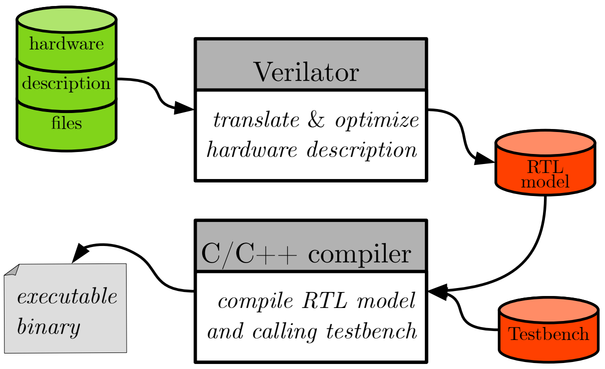


Figure 2.3.: Verilator flow
(Geier 2019)

In contrast to simulators using HDL built-in simulation techniques, Verilator performs an actual synthesis process. The output is a cycle-accurate and efficient, i.e. fast, model (Veripool n.d.). However, due to the optimization during synthesis some issues arise:

i) **Flattened hierarchy:** The output's hierarchy does not correspond the underlying hierarchical design in HDL. Verilator supports different ways to access internal variables, however, all require adding keywords in the input sources, i.e. non-functional modification to HDL sources.

ii) **Data types:** In VRTL, there are no data types representing the bit length of a variable.

iii) **Omitting non-storing elements:** Since Verilator performs a full synthesis, not all elements of combinational logic are required. Verilog nets, e.g. wires, that connect module instances in a hierarchical design are often left out. This saves execution time otherwise spent on updating signals in simulator delta-cycles.

iv) **Legibility:** HDL sources are synthesized to thousands of lines of mostly incomprehensible C++ code. This makes debugging the VRTL near impossible resulting in a "black-box" situation.

Fig. 2.3 shows a basic Verilator use case utilizing VRTL and a C++ testbench to build an executable binary for a given SystemVerilog project. Outputs are updated by stimulating input variables of the synthesized top module and calling its evaluation method. Those outputs can be handled in the hosting framework which, in this case, is a testbench instantiating the VRTL model.

2.2.2. Fault Injection

The issues resulting from Verilator's optimizations described in 2.2.1 propagate to a possible fault injection application, nonetheless. Omitted nets make soft error simulation on combinational logic impossible. Code size and legibility aggravate any manual insertion of injection points for hardware projects at complexity levels of modern processor cores. Furthermore, flattened hierarchy and the indifference of data types, both in VRTL as in Verilog/SystemVerilog, hinders locating possible injection targets in the first place.

As fault injections into wires or variables that are actually constants in hardware, e.g. look-up tables or hard-wired signals, would be unnecessary, variables in the VRTL model require classification. SystemVerilog, unfortunately, does not specify whether a declared signal is going to be continuously, i.e. a wire, or sequentially, i.e. a register, assigned at declaration time. Although this SystemVerilog supports such explicit declarations, implicit signal declarations are resolved during synthesis (IEEE Computer Society 2001, IEEE Computer Society 2009). This results in many hardware descriptions using only the signal type keyword "logic" for all signal declarations.

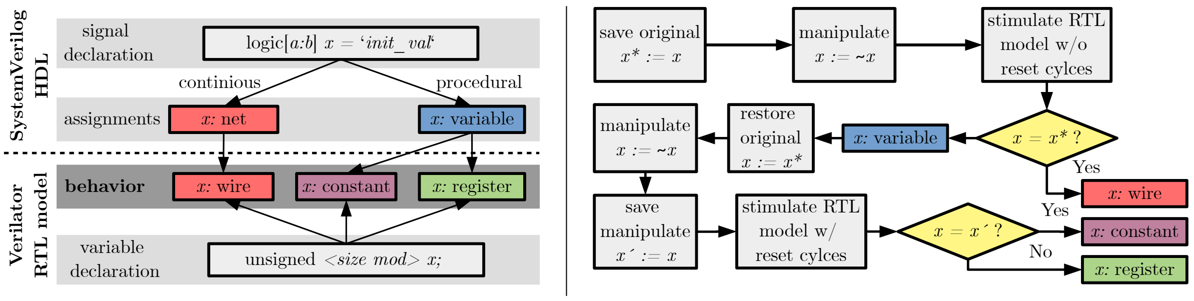


Figure 2.4.: Signal types and classification
(Geier 2019)

A possible solution to this is proposed in (Weinzierl 2017); An algorithm categorizes VRTL variables based on their behavior inside the VRTL model, which is illustrated in Fig. 2.4. On the left, the difference between a signal declaration in HDL and its corresponding variable declaration in the C++ VRTL is shown, as well as the underlying behavior. On the right, the signal classification algorithm is shown. Each input signal x is classified as either a wire, a constant, or a registers.

2.2.3. RegPicker

In (Geier 2019), the stand-alone tool RegPicker was developed which implements the classification algorithm described in sec. 2.2.2. Furthermore, the HDL's underlying hierarchy lost in the Verilator synthesis is reconstructed. Together with its classification, the signal's location is put out in an Extensible Markup Language file (XML). The XML constitutes a sufficient basis for a fault injection framework.

2.3. Simulation Target: RI5CY

This work focuses on applying the fault model described in sec. 2.1 to RTL models of cores implementing the open-hardware RISC-V ISA. Recent developments, including adaptions in commercial products, established RISC-V as an interesting topic for research.

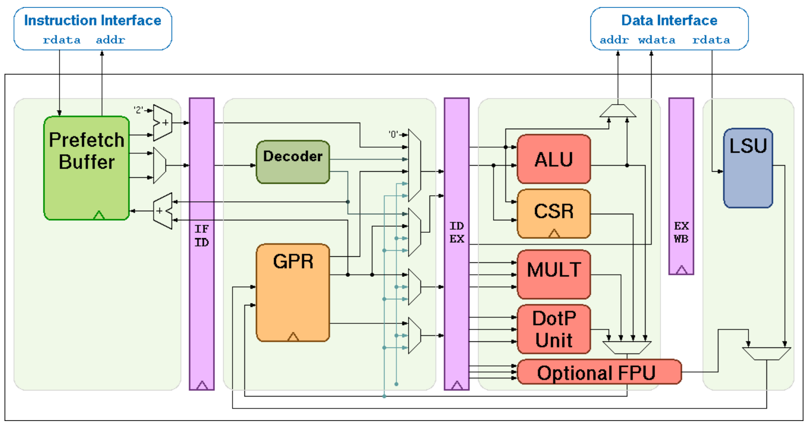


Figure 2.5.: RI5CY block diagram
(Traber et al. 2019)

As a proof-of-concept simulation target, a 32-bit core executing cross-compiled RISC-V benchmark software is chosen. RI5CY is an open-source 32-bit RISC-V derivative developed by researchers at ETH Zürich and the University of Bologna, in context of the Parallel Ultra Low Power Platform (PULP) project (ETH Zürich n.d.a, ETH Zürich n.d.b).

2.3.1. Pipeline

RI5CY has full support of RISC-V Base Integer Instruction Set, Standard Extension for Compressed Instructions, Integer Multiplication and Division Instruction Set Extension (RV32I, RV32C, RV32M), and PULP-specific ISA extensions, which are not surveyed in this work. Fig. 2.5 shows a block diagram of the 32-bit core and depicts its 4-stage pipeline:

IF Instruction Fetch. Configured to start fetching from memory address 0x80 through a 128-Bit or 4-word cache line prefetcher.

ID Instruction Decode: Interprets the fetched machine code including compressed instructions. Additionally, interrupts on illegal instructions.

EX Execute: Takes decoded instructions and schedules execution on the respective unit.

WB Write-Back: Manages storage to data memory through the Load-Store Unit (LSU) and an optional Physical Memory Protection (PMP).

2.3.2. Configuration

The RI5CY project provides either a latch-based or flip-flop-based GPR file. In this work, the flip-flop-based implementation of the register file is used and instantiated inside the core's hierarchy. Furthermore, no floating point unit is added which means the GPR's memory is reduced to the RISC-V integer register specification shown in tab. 2.1. All simulations are performed on machine privilege level, however, the privilege functionality might be affected by soft errors. The levels and their respective encoding are shown in tab. 2.2.

register	**name**	**description**
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6-7	t1-2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-11	a0-1	Function arguments/return values
x12-17	a2-7	Function arguments
x18-27	s2-11	Saved registers
x28-31	t3-6	Temporaries

Table 2.1.: RISC-V integer registers
from: (Krste Asanovic, Rimas Avizienis, Jacob Bachmeyer, Christopher F. Batten et al. 2017)

2.3.3. Injection Targets

Feeding RI5CY's HDL description into RegPicker yields 327 VRTL variables as possible injection targets comprising 5753 single-bit targets.

level	encoding	name
0	00	user / application
1	01	supervised
2	10	*reserved*
3	11	machine

Table 2.2.: RISC-V privilege levels
from: (Waterman et al. 2014)

Fig. 2.6 shows the RI5CY fault injection target cluster. The cluster is generated from RegPicker analyzing the RI5CY HDL sources and transforming the output to a directed graph description. Nodes of the graph are variables of the VRTL model. Edges between nodes, although hidden, describe a weighted hierarchical relation between variables and their parent cell, where a cell is an instance of an HDL module. Each edge's weight corresponds to the bit-length of a variable. Registers, thus injection targets, are colored green, wires and constants, thus for fault injection of no concern, colored red and purple respectively. The coordinate of each node in the force-directed graph is calculated with a converging force-layout algorithm Force Atlas in the visualization tool Gephi (Gephi n.d.). For further clarification on RI5CY's hierarchy, i.e. cell relations, fig. A.2 in the appendix shows a graph of Verilator's own XML output.

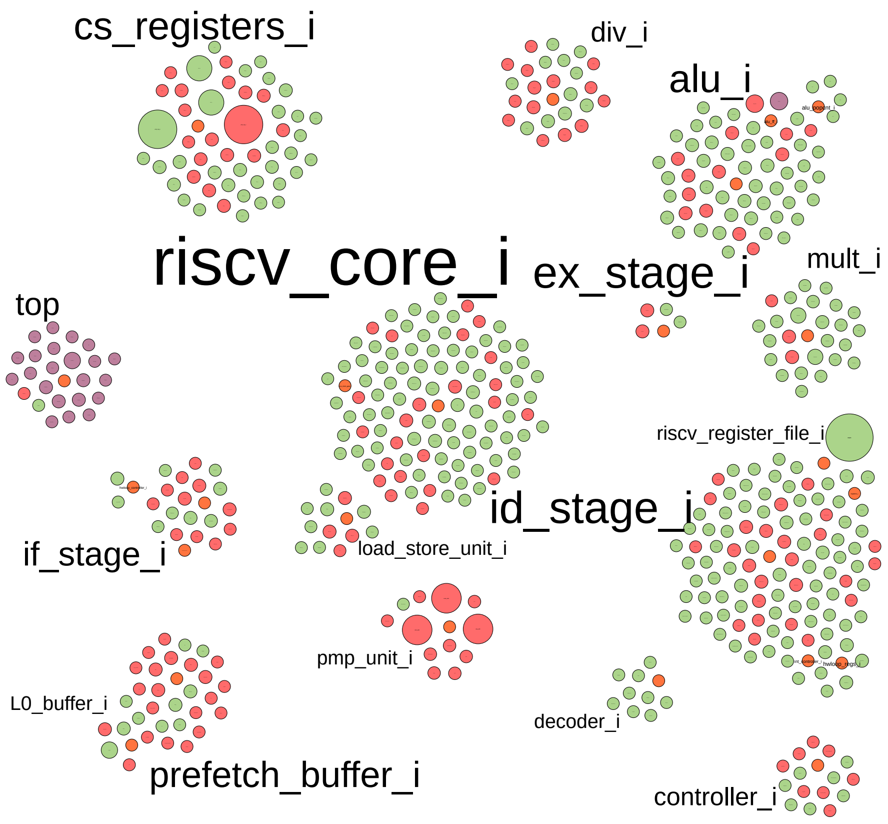


Figure 2.6.: RI5CY injection target cluster

2.4. LLVM

LLVM, formerly Low Level Virtual Machine, comprises a compartment of modular compiler and toolchain technologies (LLVM Foundation n.d.). The LLVM core libraries can be utilized to build custom tools for various applications. However, this work mostly utilizes tools revolving around LLVM's C/C++/Objective-C compiler (Clang) Abstract Syntax Tree (AST).

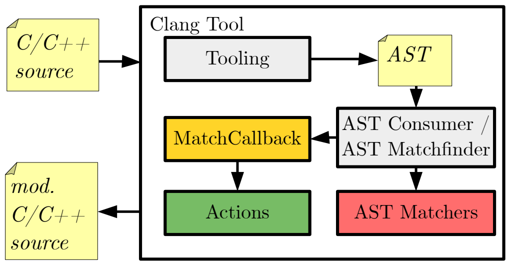


Figure 2.7.: Clang tooling

```
// clangASTexample.cpp

int main(void){

        int x   = 0;
        int X[100] = { };

        for (; x<100; ++x)
        {
                X[x] = x;
        }
}
```

Listing 2.1: Example AST source

```
FunctionDecl 0x55d1792f94c0 <clangASTexample.cpp:1:1,
|        line:7:1> line:1:5 main 'int␣()'
`-CompoundStmt 0x55d1792f9a50 <col:15, line:7:1>
  |-DeclStmt 0x55d1792f9670 <line:2:2, col:12>
  | `-VarDecl 0x55d1792f95e8 <col:2, col:11> col:6 used x
  |   |      'int' cinit
  |   `-IntegerLiteral 0x55d1792f9650 <col:11> 'int' 0
  |-DeclStmt 0x55d1792f9828 <line:3:2, col:18>
  | `-VarDecl 0x55d1792f9730 <col:2, col:17> col:6 used X
  |   |      'int␣[100]' cinit
  |   `-InitListExpr 0x55d1792f97d8 <col:15, col:17> 'int␣[100]'
  |     `-array_filler: ImplicitValueInitExpr 0x55d1792f9818
  |           <<invalid sloc>> 'int'
  `-ForStmt 0x55d1792f9a18 <line:4:2, line:6:2>
    |-BinaryOperator 0x55d1792f9898 <line:4:9, col:11> 'bool' '<'
    | |-ImplicitCastExpr 0x55d1792f9880 <col:9> 'int' <LValueToRValue>
    | | `-DeclRefExpr 0x55d1792f9840 <col:9> 'int'
    | |      lvalue Var 0x55d1792f95e8 'x' 'int'
    | `-IntegerLiteral 0x55d1792f9860 <col:11> 'int' 100
    |-UnaryOperator 0x55d1792f98d8 <col:16, col:18> 'int'
    | |      lvalue prefix '++'
    | `-DeclRefExpr 0x55d1792f98b8 <col:18> 'int'
    |         lvalue Var 0x55d1792f95e8 'x' 'int'
    `-CompoundStmt 0x55d1792f9a00 <col:20, line:6:2>
      `-BinaryOperator 0x55d1792f99e0 <line:5:3, col:10> 'int'
        |        lvalue '='
        |-ArraySubscriptExpr 0x55d1792f9988 <col:3, col:6> 'int'
        | |          lvalue
        | |-ImplicitCastExpr 0x55d1792f9958 <col:3> 'int␣*'
        | | |        <ArrayToPointerDecay>
        | | `-DeclRefExpr 0x55d1792f98f0 <col:3> 'int␣[100]'
        | |          lvalue Var 0x55d1792f9730 'X' 'int␣[100]'
        | `-ImplicitCastExpr 0x55d1792f9970 <col:5> 'int'
        |   |        <LValueToRValue>
        |   `-DeclRefExpr 0x55d1792f9910 <col:5> 'int'
        |            lvalue Var 0x55d1792f95e8 'x' 'int'
        `-ImplicitCastExpr 0x55d1792f99c8 <col:10> 'int'
          |          <LValueToRValue>
          `-DeclRefExpr 0x55d1792f99a8 <col:10> 'int'
                     lvalue Var 0x55d1792f95e8 'x' 'int'
```

Listing 2.2: Example AST dump

The AST is an intermediate representation of C/C++ source code. Listing 2.2 shows the AST for the brief example code (listing 2.1). An AST Matcher, e.g. `FunctionDecl` for declarations of functions or`ArraySubscriptExpr` for accesses to array elements, can be decorated further with more specific names to refine locating granularity. Together with Clang Tooling, the AST Matcher library enables callbacks to specific elements in the AST; Whenever the ClangTool traverses the AST and a previously defined condition is matched, a callback function is initiated with a reference to the matching AST element. This allows the user application (LLVM Frontend) to handle the code section in a desired way, e.g. analysis of transformation. In fig. 2.7 a simple Clang Tool is shown that modifies a given source file. The tooling library generates the AST which is then traversed in the AST Consumer.Actions can perform various transformations on the source file, while the AST Matchers and their respective callbacks locate the corresponding code sections.

3. Verilator RTL Modification

3.1. Verilator Evaluation

As described in Sec. 2.2.2, the VRTL model requires modifications to enable some form of fault injection methodolgy. Difficulties arising from flattened hierarchy and missing data type specifications can be handled by RegPicker (Sec. 2.2.3).

Legibility is more an indirect issue, since manually accessing variables and inserting injections inside the VRTL would not be feasible after all and the actual injections will be scheduled by an automated framework. The remaining problem is one of Verilator's greatest feats; Faster models through design optimization.

3.1.1. Synthesis

At first glance, omitted nets seem to be of no concern, since no injections are performed on combinational logic. Thus, the smallest notion of time for this VP would be one clock cycle or rather half of a clock cycle, which would comply with the fault model proposed in Sec. 2.1. However, simply changing values of internal signals of the VRTL between calls to the evaluation method has occasionally shown no effect, i.e. auto-masking the fault. Inspecting the VRTL source has shown that read-after-write occurrences for registers one stage before a module output, i.e. sequential elements driving remaining wires. Reasons can be found with Verilator's optimization of these output assignments: Since a net normally takes on the value of its most dominant driver nearly immediately, Verilator simply inserts this continuous assignment right after the driver is updated in the sequential evaluation method. This saves costly evaluation time of combinational logic, however, results in the masking issue.

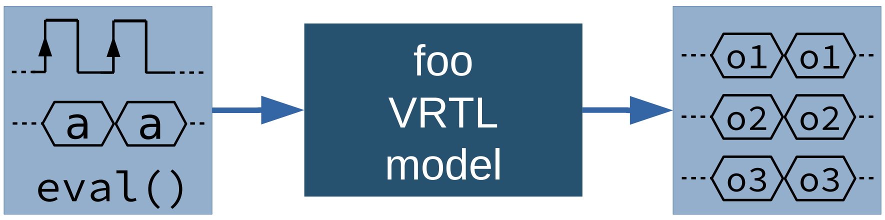


Figure 3.1.: VRTL example stimuli model

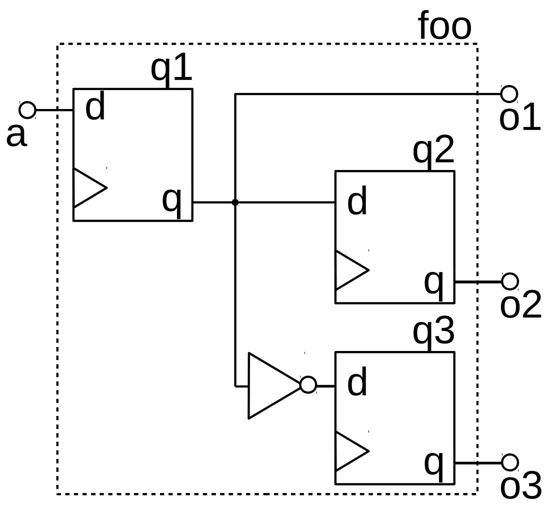


Figure 3.2.: Fault injection example

```
1  module foo(
2          input logic clk,
3          input logic a,
4          output logic o1, o2, o3
5  );
6  // declarations
7  logic q1, q2, q3;
8
9   // cont. body
10 assign o1 = q1;
11 assign o2 = q2;
12 assign o3 = q3;
13
14 always_ff @(posedge clk)
15     // seq. body
16     else
17         q1 <= a;
18         q2 <= q1;
19         q3 <= !q1;
20     end
21 endmodule
```

Listing 3.1: Example SystemVerilog

Fig. 3.2 describes a simple SystemVerilog module `foo`, the left-hand side diagram illustrates the module as a conceptual implementation. Outputs `o1`, `o2`, and `o3` are driven by `q1`, `q2`, and `q3` respectively by the `assign` statement, thus, making them equivalent to nets or wires. In general SystemVerilog provides more than one `always` keyword type supporting latch, combinational or sequential procedures. The SystemVerilog keyword `always_ff` enables procedural blocks modeling sequential logic behavior sensitive to the expression following the keyword. In this example, `q1`, `q2`, and `q3`, are sequentially updated by a non-blocking assignment (`<=`) in the clock (`clk`) -sensitive block. Non-blocking means that at sufficient sensitivity, the current state of each right-hand side non-blocking assignment is elaborated, however, the actual assignment is scheduled separately and updated at the end of the evaluation period. This means that, even though in line 17 of the listing 3.1, `q1` is apparently updated before `q2` in line 18, `q2` will be updated with the value of `q1` when the `always_ff` is entered. SystemVerilog simulators, thus Verilator, too, have to conform to these and more specifications in (IEEE Computer Society 2009).

```
VL_INLINE_OPT void Vfoo::_sequent__TOP__1(Vfoo__Syms* __restrict vlSymsp) {
    // Get internal variables from symbol table
    Vfoo* __restrict vlTOPp VL_ATTR_UNUSED = vlSymsp->TOPp;

    // Body
    vlTOPp->foo__DOT__q2 = vlTOPp->foo__DOT__q1;
    vlTOPp->foo__DOT__q3 = (1U & (~ (IData)(vlTOPp->foo__DOT__q1)));
    vlTOPp->o2 = vlTOPp->foo__DOT__q2;
    vlTOPp->o3 = vlTOPp->foo__DOT__q3;
    vlTOPp->foo__DOT__q1 = vlTOPp->a);
    vlTOPp->o1 = vlTOPp->foo__DOT__q1;
}
```

Listing 3.2: Example Verilator sequential body

Listing 3.2 shows `foo`'s VRTL (`Vfoo`) sequential evaluation method. The method is called each time a positive clock edge is recognized by `Vfoos`'s main evaluation function which is called by the simulation host. Furthermore, during one evaluation each register-like variable must not be assigned more than once, since only nets can be driven by multiple sources. Fig. 3.1 illustrates the example's stimulus, input `a` and clock signal together with a call to the main evaluation function, and outputs `o1, o2, o3`.

3.1.2. Fault Injection

Considering a fault injection between clock cycles, e.g. by accessing `Vfoo`'s variables through the VRTL symbol table `vlSymsp` the following observations can be made assuming no resets are done and clock-edges are triggered, thus, applying the fault model of sec. 2.1:

- $FI = \{\texttt{Vfoo}, \texttt{q1}, c, \texttt{flip}\}$:

  $c+1$**:** Fault propagates to `q2` and `q3` through `q1`, thus outputs `o2` and `o3` are updated with a faulty state. Injection target `q1` is updated by input `a` and `o1` is assigned the new `q1`.

  $c+2$**:** Fault has left the model.

  Problem: `o1` never sees the injected flip.

- $FI = \{\texttt{Vfoo}, \texttt{q2}, c, \texttt{flip}\}$:

  $c+1$**:** `q1` updates injection target `q2`, `o2` is assigned the new value of `q2`. Fault has left the model.

  Problem: `o2` never sees the injected flip.

- $FI = \{\texttt{Vfoo}, \texttt{q3}, c, \texttt{flip}\}$:

  $c+1$**:** `q1` updates injection target `q3`, `o3` is assigned the new value of `q3`. Fault has left the model.

  Problem: `o3` never sees the injected flip.

Consequence of this auto-masking of a section of possible injection targets is that injections between clock cycles, i.e. between evaluation calls, violates the fault model for VRTL.

3.2. Modification

Injecting the fault right after the assignment statement, would also enable updating module outputs with faulty states. Listing 3.3 shows a pseudo-modified `Vfoo` that will always result in faulty outputs. However, manually inserting these injection points in VRTL is not feasible.

```
void pseudo_inject(CData& target){
    #define c INJECTION_CLOCK
    #define b INJECTION_BIT

    // Check for injection clock cycle
    if(clock_counter == c){
        // flip target bit with bit-wise XOR
        target = target ^ (0x1 << b);
    }
}

VL_INLINE_OPT void Vfoo::_sequent__TOP__1(Vfoo__Syms* __restrict vlSymsp) {
    // Get internal variables from symbol table
    Vfoo* __restrict vlTOPp VL_ATTR_UNUSED = vlSymsp->TOPp;

    // Body
    vlTOPp->foo__DOT__q2 = vlTOPp->foo__DOT__q1;
        pseudo_inject(vlTOPp->foo__DOT__q2);
    vlTOPp->foo__DOT__q3 = (1U & (~ (IData)(vlTOPp->foo__DOT__q1)));
        pseudo_inject(vlTOPp->foo__DOT__q3);
    vlTOPp->o2 = vlTOPp->foo__DOT__q2;
    vlTOPp->o3 = vlTOPp->foo__DOT__q3;
    vlTOPp->foo__DOT__q1 = vlTOPp->a);
        pseudo_inject(vlTOPp->foo__DOT__q1);
    vlTOPp->o1 = vlTOPp->foo__DOT__q1;
}
```

Listing 3.3: Pseudo-modified example Verilator sequential body

3.2.1. Target Variables as Extended Data Structures

Concept

One solution would be to extend each targets variable data type to return the correct or faulty value on demand. Originally, Verilator synthesizes bit vectors to the smallest possible trivial C/C++ data type, i.e. size-modified `unsigned`. Substitution of the targets data-type with a class-like data-type at declaration can enable more specific handling of the element during evaluation runtime. Overloading all possible arithmetic operators and conversions generates automatic call-back functionality for each extended variable. Additionally, this would only require changing the class definitions of the VRTL modules by modifying its member's declaration types to the overloading class type.

```
 1 template <typename BaseType_t, unsigned int FIELDS>
 2 class ExtendedData {
 3 public:
 4     bool inject;
 5     BaseType_t mVar[FIELDS];
 6 protected:
 7     BaseType_t& read(int index = 0) {
 8         if (inject){
 9             ... // return injected value
10         }
11         return (mVar[index]);
12     }
13     void write(BaseType_t newVal, int index = 0) {
14         mVar[index] = newVal;
15     }
16 public:
17     inline BaseType_t& operator[](unsigned int index) {
18         if (index > FIELDS) exit(0);
19         return read(index);
20     }
21     // ASSIGNMENT Overloading
22     inline ExtendedData& operator =(const BaseType_t &driver){...}
23     inline ExtendedData& operator =(ExtendedData &driver){...}
24     // ARITHMETIC Overloading
25     inline BaseType_t& operator +(const BaseType_t &driver){...}
26     inline BaseType_t& operator -(const BaseType_t &driver){...}
27     ...
28     // CONVERSION Overloading
29     inline operator BaseType_t(){
30         return(read());
31     }
32     ExtendedData(void): inject(false), mVar(){}
33 };
```

Listing 3.4: Pseudo overloading class - extended data structure

Listing 3.4 shows a heavily shortened implementation of an extended data structure for VRTL variables. Since Verilator generates arrays of 32-bit elements for SystemVerilog variables longer than 64 bits, the actual data is stored in the `mVar` member, an array of the base or rather original data type in VRTL specified through a template argument during compilation. Besides arithmetic operators, implicit conversion (line 29) are overloaded to enable interaction with more trivial data types. The protected `read` method returns a clean or possibly faulty version of the data, while a `write` method handles correct assignment operations to the extended VRTL variable.

Evaluation

(Dittrich 2017) already utilized overloading C++ classes as extended data-types for variables in VRTL to support tracing in mixed-level simulations. While, this overloading approach allows a lot of flexibility and additional functionality, e.g. run-time tracing, simulation performance suffers.

Tab. 3.1 shows the results of a performance evaluation of the overloading approach. A VRTL model of a 32-bit RISC (OpenRisc1000) is set against a modified VRTL according to the

host execution time at 1M target cycles	GCC opt -O3	GCC opt -O0
VRTL	0.514 s	1.511 s
VRTL (modified)	8.106 s	17.861 s
slowdown	15.77	11.82

Table 3.1.: Performance overloaded VRTL

previously introduced concept. The RTL model is run for one million cycles on a modern x86 host with Linux OS. The execution time is measured in user space yielding the actual time spent inside the process hosting the VRTL models. Although not important for execution times of eventually heavily optimized simulation frameworks, a look into how a compiler handles any further modifications to the VRTL sources is interesting; Without optimizations for the Gnu Compiler Collection (GCC) C++ compiler enabled, basic VRTL is about 12 times faster than the modified one. However, for a strong level of optimization (O3), non-modified VRTL is more than 15 times faster. The assumption can be made that operator overloading is not as easily handled by the compiler with respect to code optimization as the mainly arithmetic-intensive core of VRTL is.

The basic idea of using Verilator as a tool for RTL simulation is its speed compared to other available simulators. Slowing down the simulation for fault injection would eliminate the benefits of VRTL in the first place, thus, the usage.

3.2.2. Source Transformation

Concept

While the previous approach in sec. 3.2.1 mostly relied on introducing additional functionality to VRTL variables, thus, modifying their declarations in VRTL classes, another option is to transform the VRTL's sources. The fundamental idea is to insert injection points in the model source code according to the concept introduced by listing 3.3.

host execution time at 1M target cycles	GCC opt -O3	GCC opt -O0
VRTL	2.341 s	8.165 s
VRTL (modified)	2.622 s	10.057 s
slowdown	1.12	1.23

Table 3.2.: Performance source-transformed VRTL

Evaluation

Tab. 3.2 shows the results of a performance evaluation of the source transformation concept. A VRTL model of a 32-bit RISC-V core (RI5CY) is set against source-transformed VRTL. Again, the RTL model is run for one million clock cycles on a x86 host and the simulation's execution time is measured as process user time. Without GCC optimization, original VRTL is 1.23 times faster than the modified VRTL. With optimization, the original is just 1.12 times faster. Although no injections are performed, checks for a injection at each possible injection point have to be performed which generates simulation overhead. Nonetheless, source-transformation seems to be more efficient, since most of the overhead can be resolved during compile time, where as overloading data types mostly influences the runtime.

Performance

Measuring execution time of the respective simulation process is a straight-forward way to compare the two solutions on a common host. Additionally, measuring instructions per second is commonly used and a somewhat meaningful benchmark. However, for a more independent, therefore, sophisticated, statement regarding a simulation's performance it is not sufficient. Reasons for this lay with the host. OS, implementation of the native libraries, and most of all the hardware itself can vary a lot between different simulation setups. These host-dependent influences can be reduced by comparing the number of instructions needed by the host to perform a certain number of target instructions in the simulation setup. This instruction transmission figure is still affected by the host's architecture, however, not so much by the host's other hardware, e.g. memory, cache, and CPU.

The instruction transmission measurement is setup up as followed: The simulation framework instantiates the VP. The VP executes a cross-compiled binary for a certain number of target clock cycles. In order to stimulate the VP, a clock spin is performed which is two consecutive VRTL evaluation cycles where one is with the input clock logically high and one logically low. For measuring the required host instructions, the whole setup is ran inside a GNU Debugger (GDB) process. The GDB is initiated with two breakpoints; The first where the initialization phase is done and clock spins start and the second where the program is finished. When the debugger reaches the first breakpoint, all further execution is performed step-wise, i.e. host instruction by host instruction, while the steps are counted. As an example target program a Arithmetic Logic Unit (ALU) test program is used which covers a good amount of general purpose computing in the target.

The transmission factor t_I can then be calculated with

$$i = \frac{I_{host}}{I_{target}}$$

Variable I_{host} refers to the number of host instructions and I_{target} to the number of instructions executed on the simulation target.

ALU test on x86 $I_{target} = 4,662$	I_{host}	i_{ALU}
VRTL	$115,656,447$	$24,808$
modified VRTL	$131,660,131$	$28,241$
slowdown		1.13

Table 3.3.: Instruction transmission VRTL

Tab. 3.3 lists the instruction transmission results for 1000 target clock cycles both on VRTL and source-modified VRTL with compiler optimization enabled. For the used target software an instruction transmission of $28,241$ was achieved. This means that on average, for every RISC-V instruction on the VP, about 28 thousand host instructions have to be executed. Additionally, the transmission factors should mirror the execution time slowdown effects of modified VRTL presented in tab. 3.2; the modified VRTL needs about 1.14 times more instructions than its unmodified counterpart for the same task. Additionally, since for both measurements the ALU test was used, an instructions per second (IPS) benchmark can be formulated. For example, while the host, an Intel I5 with 3.4 GHz, might yield 10 Giga-IPS, the simulation target, following i_{ALU} for modified VRTL, achieves only about 354 Kilo-IPS.

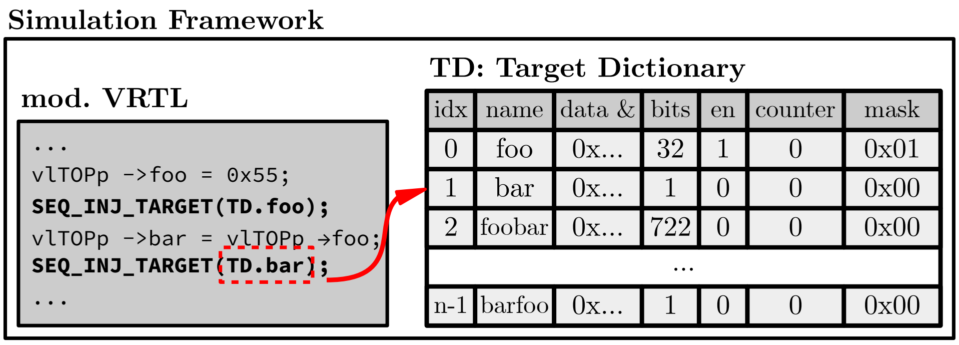
3.3. Implementation

The following sections of this chapter describes how the VRTL modifier was implemented as a stand-alone tool. For generating a RTL-custom injection framework, two major problems have to be solved by the modifier tool:

1. **API Builder:** Providing an API for managing injections in a project integrating the modified VRTL.

2. **Injection Rewriter:** Locating sequential assignments of targets in the AST and inserting injections

Both problems are basically decoupled. The only interaction happens, when an injection call accesses variables of the fault model set by the API. The API provides the user (framework) and injection call a common target dictionary holding information about each target's name, bit length, and reference to the original data element in the VRTL model. Furthermore, each target dictionary entry holds data for the injection purpose including a general injection enable, injection counter to avoid multiple injections per clock cycle, and a mask field specifying which bit of a multi-bit target has to be affected and which not.



```
...
vlTOPp ->foo = 0x55;
SEQ_INJ_TARGET(TD.foo);
vlTOPp ->bar = vlTOPp ->foo;
SEQ_INJ_TARGET(TD.bar);
...
```

idx	name	data &	bits	en	counter	mask
0	foo	0x...	32	1	0	0x01
1	bar	0x...	1	0	0	0x00
2	foobar	0x...	722	0	0	0x00
...						
n-1	barfoo	0x...	1	0	0	0x00

Figure 3.3.: Injection call with target dictionary

Fig. 3.3 illustrates the interaction between modified VRTL and the target dictionary. The VRTL calls the injection macro `SEQ_INJ_TARGET` with the respective target dictionary entry. Both, location and arbitration of the injection call have to be managed by the Injection Rewriter (sec. 3.3.2) during the source transformation.

```
#define FI_UNLIKELY(x) __builtin_expect(!!(x), 0)

#define SEQ_INJ_TARGET(TDentry) { \
  if(FI_UNLIKELY((TDentry).enable)) { \
    if(((TDentry).counter <= 0) and (TDentry).mask) { \
      *((TDentry).data) = *((TDentry).data) ^ (TDentry).mask; \
      (TDentry).counter++; \
    } \
  } \
}
```

Listing 3.5: Injection macro

Listing 3.5 contains the injection macro. Since most of the time and for the majority of all targets no injections are performed, an additional macro helps the compiler predict control flow for better simulation performance.

3.3.1. Injection API Builder

The API Builder handles generating a corresponding injection API for a given VRTL model. The information provided by RegPicker (sec. 2.2.3) suffices as input, i.e. all targets/registers, their location and bit length, etc.

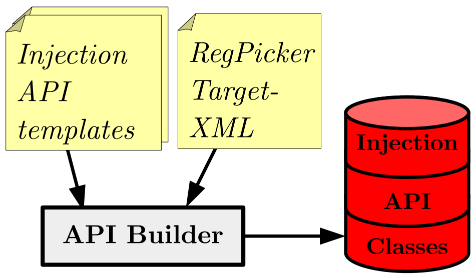


Figure 3.4.: API Builder process

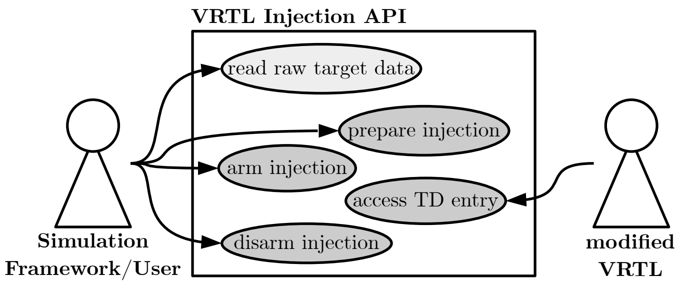


Figure 3.5.: Injection API use case diagram

Fig. 3.5 shows the use cases for the injection API. On the one hand, modified VRTL accesses the API's underlying target dictionary. On the other hand, a simulation framework or API/VRTL user, instantiating both API and modified VRTL, configures and schedules the injections, thus accessing the target dictionary. Scheduling means arming or disarming an injection at the exact injection cycle, which has to be managed by the simulation framework.

Target information is taken from RegPicker's output XML and translated into corresponding C++ classes, the target dictionary entries (Fig. 3.4). Template files contain all RTL-invariant features, e.g. the target dictionary class, injection macros and a general injection API wrapper class. The templates get extended by the modifier program with RTL-variant, or rather specific, contents mostly related to the dictionary.

The simplified class diagram in fig. 3.6 describes the relations of the injection API's different classes. A common pure-abstract base class `TDentry` acts as an interface to each specific entry

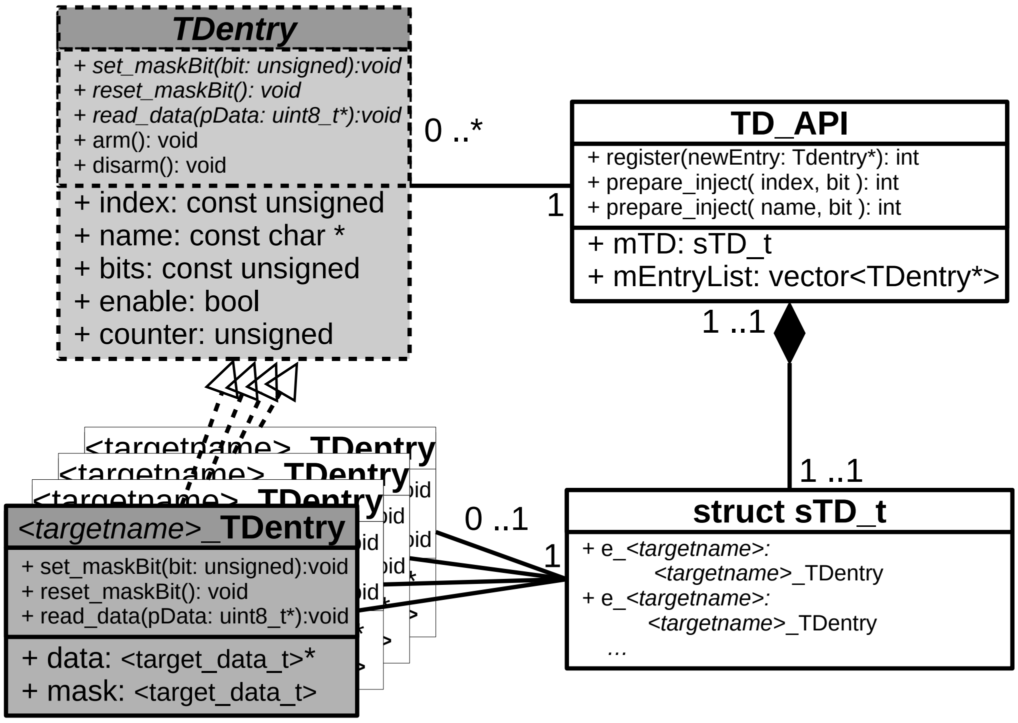

TDentry
+ set_maskBit(bit: unsigned):void
+ reset_maskBit(): void
+ read_data(pData: uint8_t*):void
+ arm(): void
+ disarm(): void
+ index: const unsigned
+ name: const char *
+ bits: const unsigned
+ enable: bool
+ counter: unsigned
0 ..*
1
TD_API
+ register(newEntry: Tdentry*): int
+ prepare_inject(index, bit): int
+ prepare_inject(name, bit): int
+ mTD: sTD_t
+ mEntryList: vector<TDentry*>
1 ..1
1 ..1
<targetname>_TDentry
+ set_maskBit(bit: unsigned):void
+ reset_maskBit(): void
+ read_data(pData: uint8_t*):void
+ data: <target_data_t>*
+ mask: <target_data_t>
0 ..1
1
struct sTD_t
+ e_<targetname>: <targetname>_TDentry
+ e_<targetname>: <targetname>_TDentry
...

Figure 3.6.: Injection API target dictionary class diagram

(`<targetname>_TDentry`), e.g. for 100 targets, there will be 100 specific entry classes. Reason for this is that now the more specific entries can still be accessed generally even though their member data, corresponding to their unique VRTL counterparts, varies. This concludes strong-typed references/pointers to the VRTL `data` element and the equivalently typed `mask`. Each target dictionary entry implements a specific `set_maskBit` method and a `read_data` method that reads the entry's data byte-wise. The target dictionary API class `TD_API` holds all entries as a vector of abstract `TDentry` pointers. The various entries get registered in this vector at the initialization phase of the simulator. The actual entry class instances are summed up in a target dictionary struct member of `TD_API`, where as the vector enables preparing and resetting injections by target name or dictionary index without the knowledge of the specific entry class.

3.3.2. Injection Rewriter

The modifier's scheme for rewriting the VRTL sources to support guaranteed injections is illustrated as an abstract control flow in fig. 3.7. The sequential injection assignments (SIA) have to be extracted from VRTL source file before they can be appended with their respective injection call.

The underlying nature of inserting or rather transforming size-wise unmanageable source files justifies the development of an LLVM-based tool for this VRTL modification.

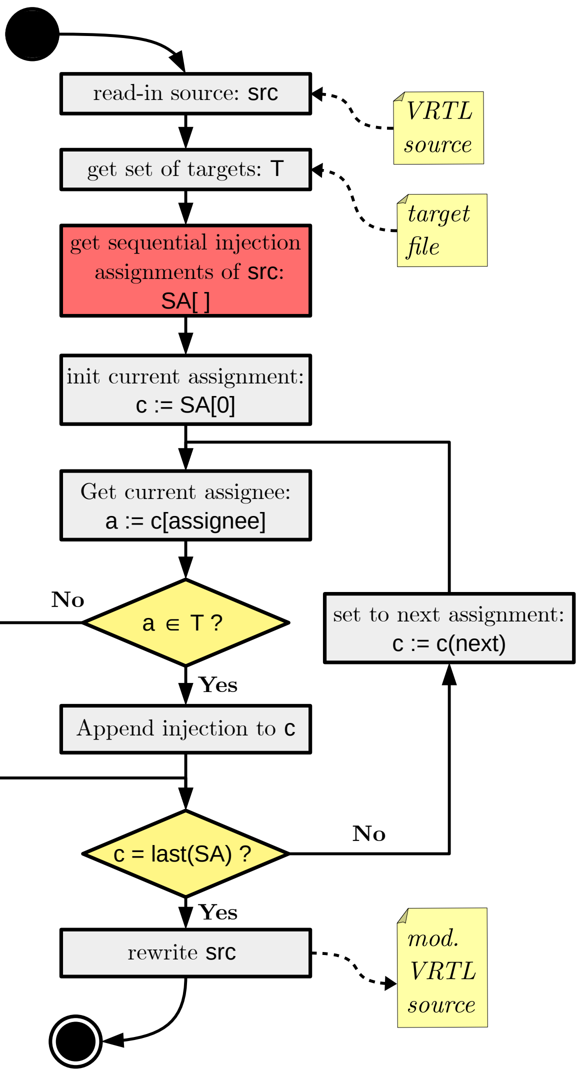

read-in source: src
VRTL source
get set of targets: T
target file
get sequential injection assignments of src: SA[ ]
init current assignment: c := SA[0]
Get current assignee: a := c[assignee]
No
a ∈ T ?
Yes
set to next assignment: c := c(next)
Append injection to c
c = last(SA) ?
No
Yes
rewrite src
mod. VRTL source

Figure 3.7.: VRTL source transformation algorithm

Extracting Sequential Injection Assignments

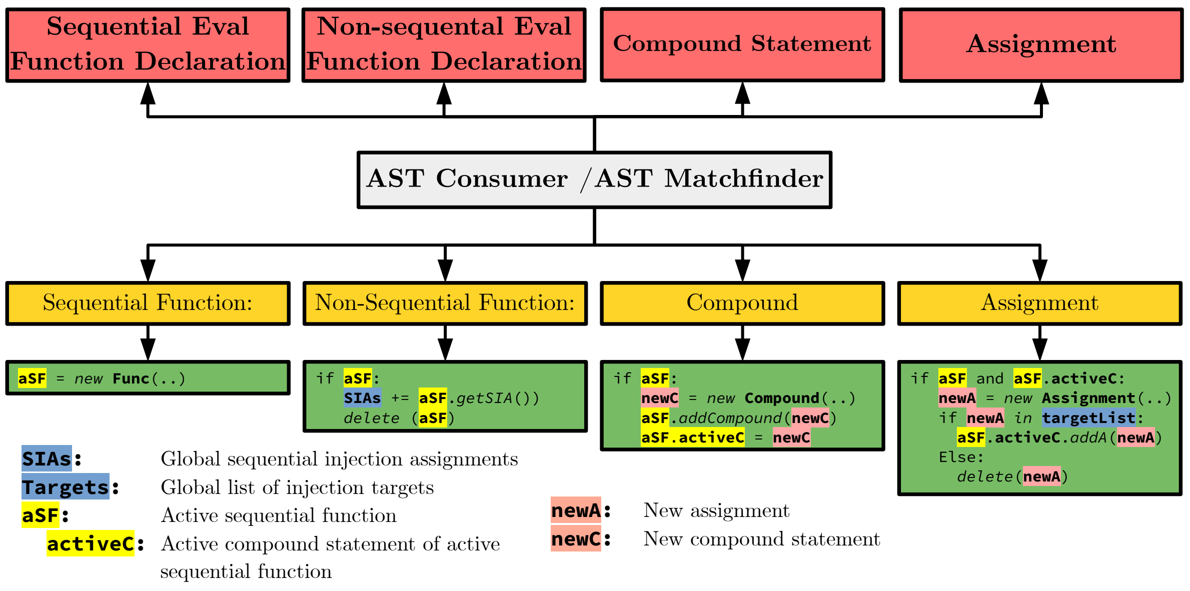
Utilizing the AST Matchfinder to efficiently extract sequential assignments that are of interest for injection requires analyzing the properties of such assignments:

i) The assignee is a target identified beforehand, e.g. by RegPicker,

ii) the assignment is located inside a sequential evaluation method,

iii) the assignment is the last inside one compound, i.e. basic block.

While i) and ii) are straight-forward essential properties, iii) traces back to the source's language boundaries. In C/C++ the largest possible bit-size of a variable is capped, e.g. 64 bit. However, in HDLs vectors of much larger bit-size can be specified. As described in sec. 3.2.1, Verilator handles this by specifying arrays where each field has up to 32-bit and essentially breaks up assignments into multiple field-individual ones executed one after another in the same compound. In addition, when variables have to be accessed bit-wise, i.e. not the full bit-vector is assigned a new value, Verilator synthesizes this similarly to multiple field access, however, for a single field in the same compound. Inserting the injection macro at the first occurrence of a target's assignment in the same compound could then lead to a self-masking process, i.e. an injection-following assignment might overwrite the injection. This issue is engaged by ensuring only the last, or rather, dominant, assignment for a single target in a compound statement has an injection point.


Sequential Eval Function Declaration
Non-sequental Eval Function Declaration
Compound Statement
Assignment
AST Consumer /AST Matchfinder
Sequential Function:
Non-Sequential Function:
Compound
Assignment

```
aSF = new Func(..)
```

```
if aSF:
   SIAs += aSF.getSIA())
   delete (aSF)
```

```
if aSF:
   newC = new Compound(..)
   aSF.addCompound(newC)
   aSF.activeC = newC
```

```
if aSF and aSF.activeC:
   newA = new Assignment(..)
   if newA in targetList:
     aSF.activeC.addA(newA)
   Else:
     delete(newA)
```

SIAs:	Global sequential injection assignments
Targets:	Global list of injection targets
aSF:	Active sequential function
activeC:	Active compound statement of active sequential function
newA:	New assignment
newC:	New compound statement

Figure 3.8.: LLVM Tool - Finding sequential injection assignments

Fig. 3.8 shows a simplified interpretation of the LLVM/Clang AST Matcher-Callback setup for SIAs in the VRTL sources. There are four concurrently running AST Matcher-Callback pairs in use:

- **Sequential Eval Function Declaration:** Matches function declarations that correspond to VRTL sequent function declarations. On callback, a new active sequential function (aSF) ob is generated.

- **Compound:** Matches all braced statements. If an active sequential function is found, the new compound object is added to the function object. The function objects saves the last found compound statement (active compound).

- **Assignment:** Matches all assignments. If the new assignment is found in the previously defined target list, it gets added to the active sequential function object's active compound statement.

- **Non-Sequential Function:** Matches all miscellaneous function declarations, i.e. non-sequential function. If a sequential function is active when this matcher is raised, the AST traversal left this aSF's declaration's body. Therefore, the aSF was fully analyzed and its SIAs can be extracted. The extraction iterates over all compounds returning only the dominant sequential assignments. Afterwards, the aSF gets discarded which avoids further assignments or compounds being added.

4. Simulation

4.1. Simulation Framework

The simulation framework manages the modified VRTL core, an injection scheduler, and a user interface (UI). Additionally, a reduced virtual SoC (VSoC) is modeled. For better simulation performance, the framework is implemented in C++, which allows an efficient executable. Fig. 4.1 shows the framework's components and structure.

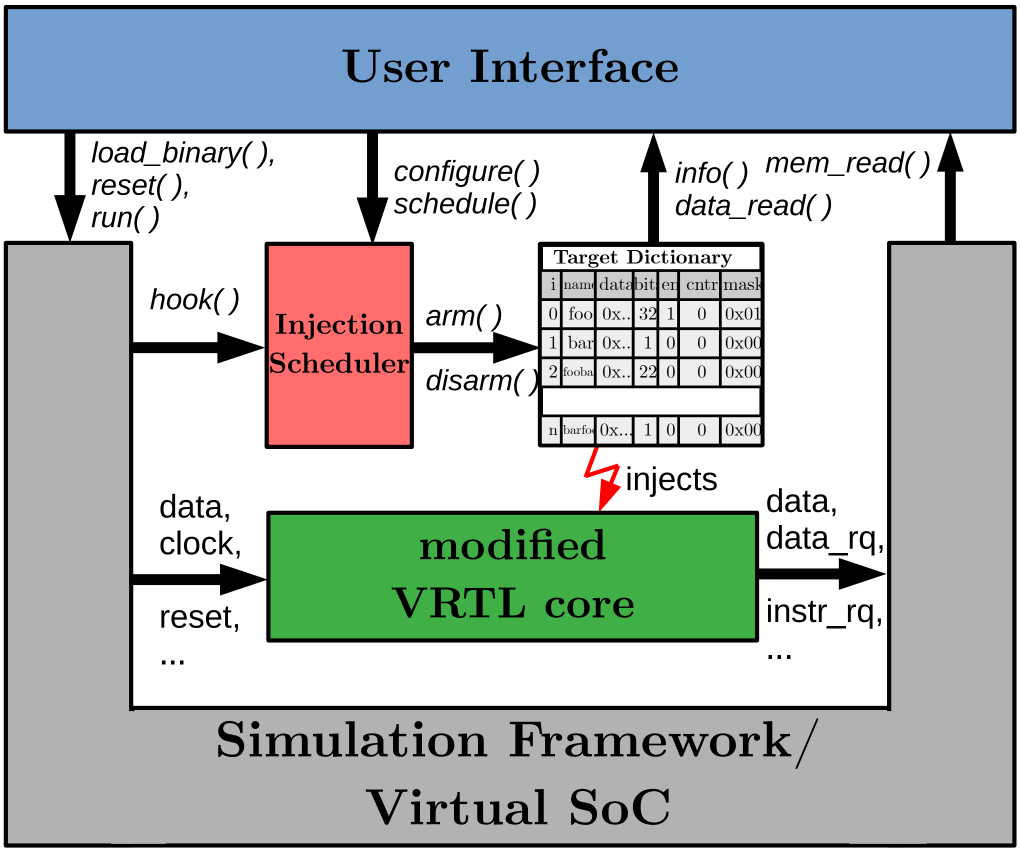


Figure 4.1.: Simulation framework

4.1.1. Modified VRTL Core

The modified VRTL core, as described in section 3.2, is integrated in the simulation as generated by the modification tool. Further modifications would break the abstraction layer generated by the automated design flow.

4.1.2. Reduced Virtual System on Chip

The simulation framework also mocks a reduced VSoC for the VRTL core. Although Verilator enables a SystemC model for better transaction-level modeling, the C++ output is used as the focus of this work is on the RISC cores and not on any additional peripherals. Therefore, the VSoC has to provide only the memory for the core to interface with. Since RI5CY has a data memory interface and an instruction memory interface, both have to be modeled by the VSoC and their transactions forwarded to the virtual memory.

Interface to VRTL

In short, the following functionality has to be provided by the virtual SoC component to the VRTL model:

i) **Clock spin:** Stimulate with clock clock cycles.

ii) **Control:** Set control outputs, e.g. enable instruction fetch, reset, ...

iii) **Instruction memory:** Provide instruction fetch with instruction data in the virtual memory.

iv) **Data memory:** Support load and store access to the virtual data memory.

Interface to Simulation Framework

Additionally, these tasks issued by the simulation framework have to be handled:

i) **Boot:** Load a cross-compiled binary to the virtual instruction memory and initliaze control signals.

ii) **Execute:** Run the model for a certain amount of clock cycles.

iii) **Reset:** Set the model to a pre-defined state.

iv) **Injection hook:** Before evaluating a new clock cycle a possible injection scheduler hook function is called.

The simulation framework poses the first instance of manual input for each new RTL model in the proposed simulation toolchain. The reason is that it has an interface with the VRTL core, which is not standardized. For example, the examined RI5CY RISC-V core has a slim input/output profile assuming that more complex interfaces are implemented on top of it, e.g. connect the memory interfaces with a on-chip bus. On the other hand, cores like the OpenRisc1000 implement an on-chip wishbone-bus interface on the RTL's top module.

4.1.3. Injection Scheduler

The injection scheduler block acts as an interface to the target dictionary API generated by the modification stage. Additionally, a hook method can be called by the main clock-spin evaluation method in the framework, thus, allowing the scheduler to decide whether an injection should be performed in the current cycle. Injections can be scheduled by passing the target's index or name in the dictionary to the scheduler. Additionally, either the exact bit and clock cycle can be specified or both selected randomly. For random bit, a pseudo random number generator (PRNG) selects a bit within the bit-length of the target. For random clock cycle, the PRNG generates a random number within a configurable interval, e.g. before the faulty simulation, a reference run could be used to determine the actual program cycle count, thus, an injection-effective time interval.

```
void RV32_SoCFrame::
clockspin(uint64_t cycles) {
  for (; cycles; cycles--) {
    //Call injection scheduler hook
    TIS->hook();

    //Low clock evaluate
    mCore->clk_i = 0;
    mInst_I->update(mCore->clk_i);
    mData_I->update(mCore->clk_i);
    mCore->eval();

    //High clock evaluate
    mCore->clk_i = 1;
    mInst_I->update(mCore->clk_i);
    mData_I->update(mCore->clk_i);
    mCore->eval();

    //Increment cycle count
    mPU->mCycles++;
  }
}
```

Listing 4.1: Virtual SoC clock-spin

```
void TargetInjectScheduler::
hook(void){

  if(mInjListSet.size()<=0) return

  if(mPU->mCycles <
    mInjListSet[0]->mInjectionCycle) return;
  else
  {
    if(PU->mCycles <=
      mInjListSet[0]->mInjectionCycle)
    {
      mInjListSet[0]->arm();
    }
    else
    {
      mInjListSet[0]->disarm();
      mInjListDone.push_back(mInjListSet[0]);
      mInjListSet.erase(mInjListSet.begin());
    }
  }
}
```

Listing 4.2: Injection scheduler hook

For each simulation run multiple injections can be scheduled. Each injections in stored in a buffer sorted for earliest first, i.e. lowest clock cycle injection first. Listing 4.1 shows the simulations clock-spin method calling the injection scheduler hook in line 5. Inside the hook, checks are performed whether any injections are scheduled (line 3) and the current clock cycle (`mPU->mCycles`) is an injection cycle (line 5, 6, 9 and 10). If an injection cycle is reached, the injection is armed meaning the enable flag in the respective target dictionary gets set. On the next call to the hook function the injection gets disarmed, deleted from the todo-list (`mInjListSet`) and appended to a done-list `mInjListDone`. Considering the hook method being called in every clock cycle, the sorted list ensures better performance, since no iterations over the injection list have to be performed.

4.1.4. User Interface

For individual handling of the simulation framework, a UI provides dynamic configuration and execution. The GDB inspired command line interface supports the inputs listed in tab. 4.1. The inputs are grouped in *help*, *soc*, *mem*, and *inject*;

help displays interface help for all groups or a specific one,

soc includes commands controlling the simulation target,

mem lets the user read virtual memory content, and hash the virtual memory,

inject can return information regarding the target dictionary and scheduled injections. Additionally, the interval for random injections can be set. The load and store functions enable reproducing injection schedules, e.g. a previously stored configuration can be reloaded from configuration file. Injections can be added explicitly or randomly. Furthermore, a hash function performs a hash over all data elements registered in the target dictionary.

On *soc run* commands the UI returns the following:

- **Application finished:** The application finished its execution. This is detected by a return from the target software's main function
- **Cycles done:** The model was executed for the set amount of cycles without detecting any system errors.
- **System level fault:** The system detected an exception, e.g. an illegal instruction.
- **Core hang-up:** A maximum amount of clock cycles was reached without finishing the program or set amount of clock cycles.

group	**command**	**required**	**options**	**example**
help				help
soc	run		[cycles: int]	soc run 100
	load	<binary>		soc load ./foo.bin
	reset			soc reset
	info			soc info
mem	read	<address>	[bytes:=4]	mem read 0xFF 4
	hash			mem hash
inject	info		[option: {schedule, targets}]	inject info schedule
	load	<config file>		inject load ./config.txt
	store	<config file>		inject store ./config.txt
	interval	<L>:<U>		inject interval 0:100
	hash			inject hash
	add		[target] [bit] [cycle]	inject add

Table 4.1.: User interface input table

4.2. Simulation Setup

The simulation setup can be partitioned in a simulation run and a classification of its results. Both are performed by individual Python scripts. To gather meaningful information about soft error effects on the core under test a large number of random injections have to be managed. The sum of all individual simulation runs is called an evaluation.

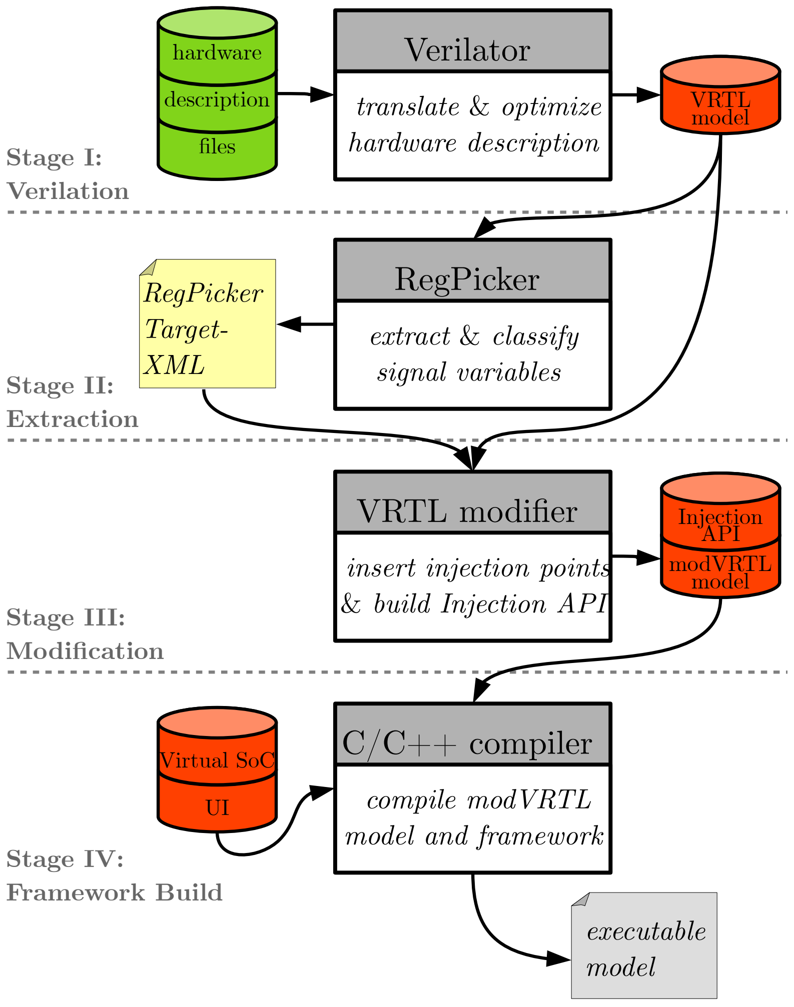


Figure 4.2.: Simulation framework toolchain

4.2.1. Target Software

An application program has to be generated. This is done by invoking the GNU RISC-V GCC for the target architecture (RISC-V n.d.). A custom linker script ensures correct linkage with the VSoCs memory space. Furthermore, a simple assembly startup file defines correct program boot behavior and exception handling. The resulting cross-compiled binary can then be passed

eval.	req. clocks	executed instructions	data memory reads	data memory writes	instruction fetches
ALU	5,904	4,658	1,823	383	1,393
MUL	16,528	11,706	4,813	1,610	3,443
AES	142,355	106,101	34,737	12,205	33,985

Table 4.2.: Target software test properties

through the UI to the simulation framework.

The bare-metal code is based on RISC-V test suits provided by the PULP project. The tests normally confirm whether a core implementing the RISC-V ISA operates as expected. Besides checking the basic functionality of the ISA, each test frequently accesses data memory, where its results are stored. Tab. 4.2 lists some properties, such as read and write accesses to memory, of the tests under normal circumstances, i.e. neither errors nor injections. Evaluation of the RI5CY core was done for an Arithmetic Logic Unit (ALU) test, a Multiplier (MUL) test, and an Advanced Encryption Standard (AES) test. In used prefetch buffer configuration, RI5CY reads 128-bit of instruction data per fetch from memory, which translates to four non-compressed instructions.

4.2.2. Evaluation

To generate a multitude of simulation results as fast as possible, a cluster of linux host machines is used. Each host runs a Python script interacting with the simulation framework's UI by opening a pipelined sub-process. The individual result of a simulation run is appended to a host-unique result file in a simulation setup common directory on the server.

Fig. 4.3 shows the control flow and result for a complete evaluation. The simulation run is initialized with the executable model from the simulation framework toolchain's output and a cross-compiled application software. Listing 4.3 shows an example for a single simulation result returned by the framework's UI. For clarification, the UI commands invoked by the python script are inserted as green lines in the listing and output from the UI in purple. Each simulation run gets a simple header with an index and time stamp determining its start and identity. At first, the framework gets passed the cross-compiled binary's file-location which initiates the VSoC (line 5). Then, the injection interval is set to a significant value, e.g. the expected executing cycles, and a random injection is added and read back (lines 7-9). The simulation is now set up and a endless run invoked (line 13). When the scheduler hook injects the fault, the action is reported by the UI (line 14). In this example, the application software's main function returns to the startup files call to `main()`, which indicates its finishing state in line 15. Finally, internal information of the VSoC is read out (line 16). This includes the total number of clock cycles and instructions, a hash over the data and instruction memory, a hash over all target dictionary entries' data, and two memory access flags. The memory access flags indicate whether the load script-defined address spaces were violated, i.e. a write to the

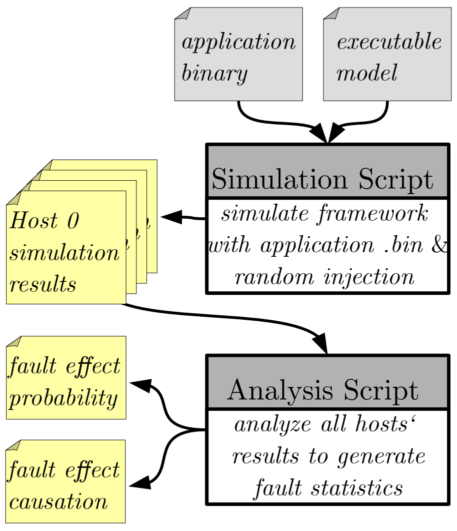

application binary
executable model
Simulation Script
simulate framework with application .bin & random injection
Host 0 simulation results
Analysis Script
analyze all hosts‘ results to generate fault statistics
fault effect probability
fault effect causation

Figure 4.3.: Evaluation run

```

29 NEW_LOG: 18/02/2020 15:49:33
###########################################################
#         Ri5cy SoCFrame command line UI v0.9             #
###########################################################
// soc load application.bin
#[CORE] Core settings: PULP_SECURE = 1, N_PMP_ENTRIES =
     16, N_PMP_CFG 4
// inject interval 0:5904
// inject add
// inject info schedule
#TODO:
#LEGEND: (index): target-name[bits] - <bit> at <cycle>
#(318): TOP.top.
     __PVT__riscv_core_i_cs_registers_i_pmp_reg_q[768] -
     530 at 3016
// soc run
#INJECT (3016): TOP.top.
     _riscv_core_i_cs_registers_i_pmp_reg_q armed.
#RESULT: Application finished.
// soc info
#INSTRUCTIONS: 4658
#CLOCK_CYCLES: 5904
#DATAMEM_HASH: fa055b23cd7d95a1
#INSTMEM_HASH: 9bd1eef2d276b31f
#TARGET_HASH:  925a24adcb9dc587
#DATAMEM_ACC:  1
#INSTRMEM_ACC: 1
```

Listing 4.3: Simulation run output

instruction ram was performed or a read from an invalid address.

The memory hashes are used to reduce the amount of data needed to store each simulation result of an evaluation. Due to hashing, information is lost and a fine-grained statement about a soft-errors effect is harder to make. However, fault effect classification is still possible, since mostly a general effect on the system and not on the individual application is investigated.

5. Evaluation

5.1. Error Classification

The individual simulation results of one evaluation are classified according to the fault effects listed in sec. 2.1.3. The errors are prioritized such that a more severe core hang-up that also generated a memory corruption is declared a hang-up fault and not an application level fault. Tab. 5.1 shows the possible fault effects and their priority level. Fault effects on the same priority level can overlap, e.g. a simulation run can be classified both as an Application Output Error (AO) and an Application Program Flow Error (AP).

key	**priority**	**type**
HP	0	Hang Error - Pipeline Stall
HL	1	Hang Error - Logic Stall
SI	2	System Error - Illegal Instruction
SM	3	System Error - Memory Access
AO	4	Application Error - Output
AP	4	Application Error - Program Flow
NSK	5	Soft Error Not Masked
MSK	6	Soft Error Masked

Table 5.1.: Fault effect priority levels

Fig. 5.1 illustrates the classification process. After the simulation result, including execution result, memory hashes, instruction count and total clock cycles, is read, a check is performed whether a hang-up was detected. If additionally, the maximum instruction count was reached, i.e. instructions were still executed, the fault is declared a logic hang-up. This means, the core still executed instructions, however, no program termination was accomplished. In reverse, i.e. the maximum instruction count was not reached, a pipeline hang-up occurred. If the simulation returned a system error, either an illegal memory access or an illegal instruction was detected. The next priority level involves application level faults. Here, output corruption is checked by comparing the memory hash value with an expected one. Additionally, the program flow could have been corrupted which is checked by comparing the simulation's instruction and cycle count. At last, meaning the application must have finished without errors, the algorithm checks the hashed injection target data for deviations to an error-free run. No deviation means that the soft-error was fully masked.

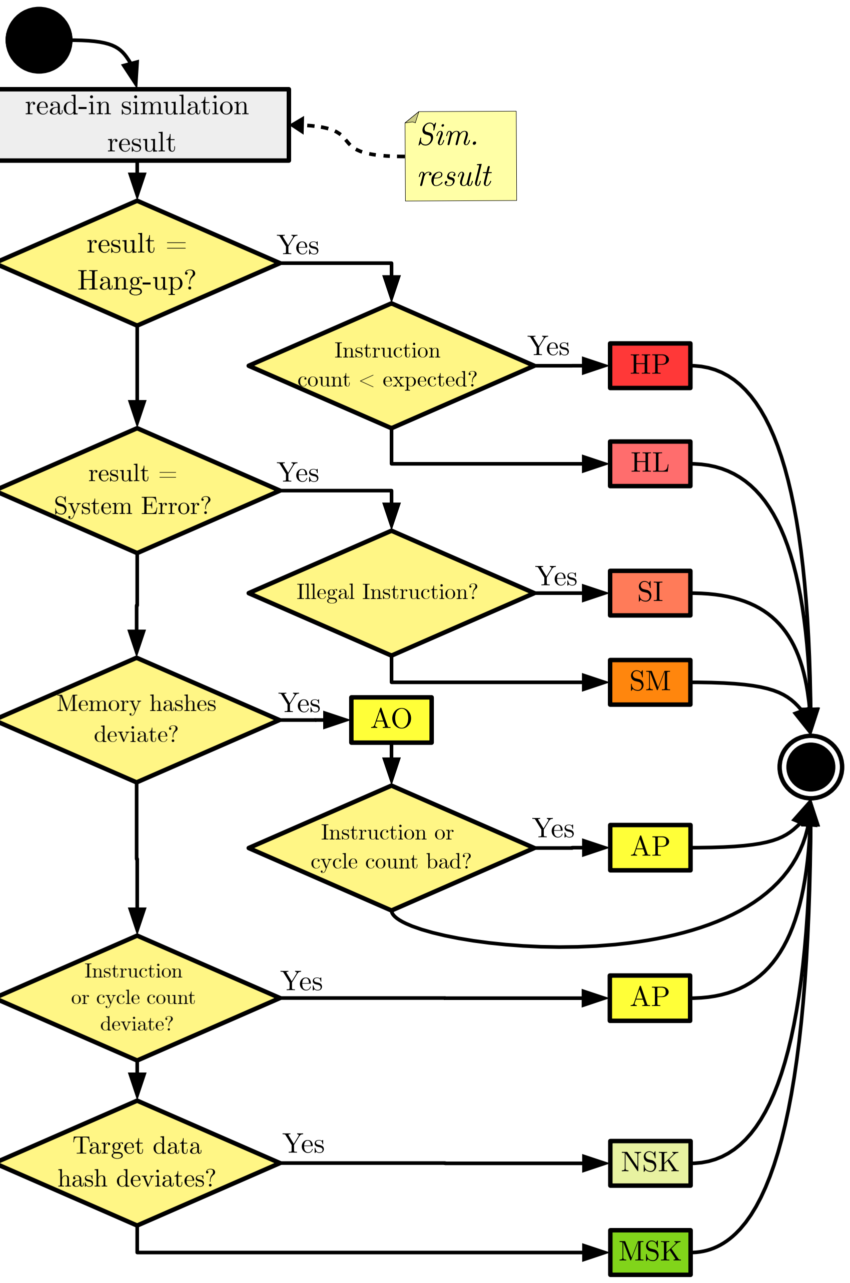

read-in simulation result
Sim. result
result = Hang-up?
Yes
Instruction count < expected?
Yes
HP
HL
result = System Error?
Yes
Illegal Instruction?
Yes
SI
SM
Memory hashes deviate?
Yes
AO
Instruction or cycle count bad?
Yes
AP
Instruction or cycle count deviate?
Yes
AP
Target data hash deviates?
Yes
NSK
MSK

Figure 5.1.: Fault effect classification

5.2. Results Analysis

There are different ways and dimensions to analyze the simulation results. Since injections are randomized in space (bits) and time (clock cycle), relations between fault effects and both dimensions can be formulated. In this work's proof-of-concept evaluations, however, the focus is on the space dependency of specific fault effects with only a small notion to individual time dependencies. Information gathered from fault effect classification of all simulation runs in the evaluation is used to observe the following relations:

i) **Fault effect probability:** Assuming that every injection target bit has an equal probability being affected by a soft-error, what is the probability of a certain fault effect occurring. Furthermore, the effect that different application code has on the effect is examined.

ii) **Fault effect causation:** Assuming that a certain fault effect occurred, what is the probability that a soft-error in an specific injection target was the cause. Additionally, with a large enough statistical sample size, time dependencies of certain fault effect causes can be observed.

5.2.1. Statistical Fault Effect Probability

For each evaluation (ALU, MUL, AES) a finite number of unique injections exist, i.e. the initial population N. This number can be calculated by multiplying the vector of all target bits with the vector of the individual application execution cycles. For the evaluation-common target RI5CY, RegPicker identified 327 target registers making up 5753 individual target bits. For the ALU evaluation, this results in an initial population N_{ALU} of $33.97e6$. For MUL and AES, $95.09e6$ and $818.97e6$ respectively. These huge population sizes mean that a verification-level evaluation, i.e. all possible simulations are evaluated, would take a very long time. For example, verifying the AES test would take $818.97e6 \times 142,355$ total simulation cycles on the model or $818.97e6 \times 106,101$ instructions under perfect conditions, e.g. no evaluation overhead or simulations with time-outs. Assuming the estimated IPS of $354e3$ (see sec. 3.2.2) for the modified VRTL, this would result in a total $245e6$ seconds, or 7.8 years, of simulation time for a single-core simulation process. Obviously, not a worthwhile figure.

(Leveugle et al. 2009) introduced a way how a minimum sample size n for a given initial population N, a desired margin of error e and confidence t (cut-off point with respect to a normal distribution), can be calculated. The prerequisites are that n is uniformally distributed over N and the probability of a specific fault effect p is unknown.

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \tag{5.1}$$

Equation 5.1 shows (Leveugle et al. 2009)'s formula to calculate the sample size. Since p is unknown and the most conservative approach is a large sample size, the highest n is gained by assuming $p = 0.5$ regardless of a rational probability of an effect. Furthermore, eq. 5.1 can be converted to calculate a margin of error for a given confidence level, sample size, and initial population:

$$e = t \times \sqrt{\frac{p \times (1-p)}{n} \times \frac{N-n}{N-1}} \tag{5.2}$$

For each evaluation of the RI5CY core, the sample size varied. For example, the ALU test was simulated about 4.6 million times. Assuming each bit and cycle has an equal probability being selected for injection, at 5753 injection bits and approximately 6000 simulation clock cycles about one seventh of all possible injections were evaluated. The uniformity of the injections is illustrated in fig. 5.2 for the ALU test. Each dot represents a single simulation with its target bit displayed on the vertical axis and the injection cycle on the horizontal axis.

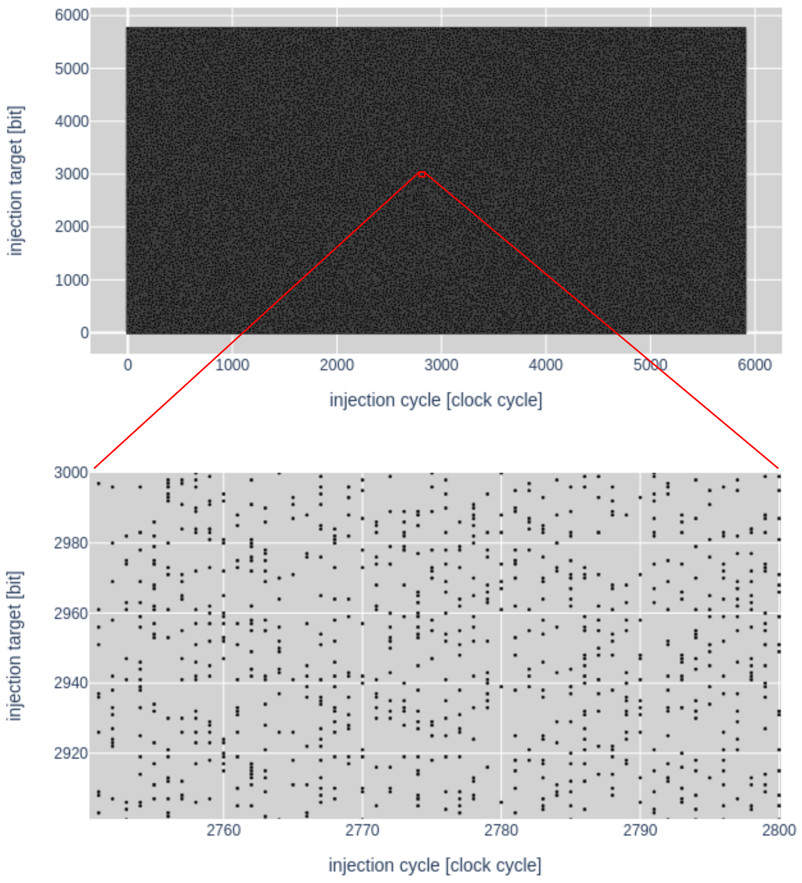


Figure 5.2.: Evaluation cluster

With eq. 5.2, each evaluation's fault effect probability results can be given with the addition that this result is within a margin of error at a fixed confidence level. Applied to the statistical fault injection proposed by (Leveugle et al. 2009), this results in the error margins listed in tab 5.2 for the various evaluations done on the RI5CY core. For example, for the ALU test a probability of 3.49% for a random injection resulting in an application level fault was observed. This figure can now be said to be with a confidence of 99.8% within the unknown correct probability for application level faults and an error margin of 0.0663%. For the AES test, which has a large population size and small sample size with respect to the other two tests, a higher margin of error has to be considered when formulating statements regarding fault effect probabilities. The error margins should be kept in mind when evaluating the statistical results for all tests.

eval.	initial population N	sample size n	$t = 1.96$ 95% conf.	$t = 2.5758$ 99% conf.	$t = 3.0902$ 99.8% conf.
ALU	33,965,712	4,683,165	$e = 0.0420\%$	$e = 0.0553\%$	$e = 0.0663\%$
MUL	95,085,584	3,704,735	$e = 0.0499\%$	$e = 0.0656\%$	$e = 0.0787\%$
AES	818,968,315	2,590,085	$e = 0.0608\%$	$e = 0.0799\%$	$e = 0.0959\%$

Table 5.2.: Evaluation error margins for given confidence levels

5.2.2. Data Processing and Visualization

Although a confident statement about error probabilities can be formulated with a relatively small sample size, interpreting effect causes requires looking at more than just the end result of a simulation run. To evaluate relations between injection space and time more easily, the evaluation data is further processed and visualized.

Causation Clusters

The evaluation results are analyzed and fault effect occurrences counted. To examine fault causation, the effect is set in relation to a specific injection target (space) being the initial cause for deviations from the golden reference model. The resulting statistic yields a space dependent probability for faults. For better interpretation of the individual fault effect causes, each multi-bit target is one possible cause. This has the consequence, that a large target, e.g. the register files memory element with 1024 bits, will naturally have a larger impact during the evaluation due to its inherent size, thus, occurrence ratio in random injections. Therefore, two analysis cases are taken into account:

a) **Unweighted causation statistic:** A large target might have an inherently higher impact factor

b) **Weighted causation statistic:** All causes are inversely weighted with their size, i.e. the smaller the target, the higher is its impact factor

For visualization purposes, the cause statistics are merged with the models hierarchy and formed to a force-directed graph analog sec. 2.3.3. Fig. A.1, showing the RI5CY target cluster, can be used as a reference graph identifying the RISC-V core's components and its purple-colored injection targets. The visualization tool Gephi takes data in XML-like formats. For this use case, the RegPicker XML file was transformed to a node/edge representation and each target, i.e. node, is decorated with its proportional share in the various fault effects as XML attributes. The tool can be configured to display node colors and their label size in relation to one of its attributes' value. For example, all nodes have an attribute *signalClass* according to RegPicker's classification and the tool can color each node depending on this attributes' value, e.g. green for *wires* and purple for *registers*. Similarly, all nodes can be colored depending on their respective proportional cause for a fault effect, e.g. a hang-up.

Heat Maps

Furthermore, execution time dependency can be evaluated, e.g. does a target always result in a specific effect or only at particular instants in the execution. For this, the sparse evaluation sample (matrix $\tilde{S}$) is mapped to a more coarse-grained, thus dense, representation of the initial population (matrix $\tilde{Z}$). The matrix rows N represent the injection space (bits) and the matrix columns M the injection time (execution cycles). To allow better comparisons between different evaluations, the matrix size ($M \times N$) is constant for all evaluations. Each element $z_{m,n}$ holds a value calculated from the occurrences of a specific fault effect resulting from a injection in the respective sector (multiplicities of unique simulation parameters are filtered out). For example, the ALU test normally has 5904 execution cycles in the reference model and 5753 injection bits. Since not all possible simulations (5904×5753) are performed, each simulation run is mapped to a sector of the initial population size, i.e. an element of $\tilde{Z}$. For each individual fault effect, one $\tilde{Z}$-matrix is generated and all evaluation results mapped to the respective matrix and its element. The array of $\tilde{Z}$-matrices is referred to as ζ and its construction is described in the pseudo code algorithm 1. Each $\tilde{Z}$-matrix of ζ is initialized with some form of invalid or Not-a-Number (NaN) value. Reason is that this allows distinguishing sectors not creating a specific effect from ones where no information regarding an effect is available. In other words, a sector where no data is available must not be interpreted as fault-free.

The $\tilde{Z}$-matrices can be visualized by so called heat maps. Fig. 5.3 shows the heat map gained from mapping the evaluation sample and all results of erroneous fault effects of the ALU test to a (1000×250)-$\tilde{Z}$-matrix. Therefore, one element of $\tilde{Z}$ covers a sector of 6×23 cycles and bits respectively. Erroneous fault effects refer to all fault effect classes excluding NSK and MSK. Dark blue coloration means the section mostly results in masking effects, while a bright yellow means high amounts of error effects. The pattern allows to identify vulnerable injection targets and also their time dependency. For example, a bright continuous line could mean, that a target or bit space almost always results in a fault, while a discontinuous one points to a error-dependency towards certain stages in the model's execution. The heat map is not intended to actually determine which injection bit and which cycle was responsible for a fault. However, this type of visualization can be a starting point for more fine-grained inspections.

Algorithm 1: Mapping sparse evaluation results to dense effect matrices

```
function Z-Map ($N, M, B, C, R, F$);
Input  : Maximum number of rows $N$ and columns $M$,
         sum of all injection bits $B$ and maximum injection cycle: $C$,
         set of all fault injection evaluation results: $R$, set of all fault effects: $F$
Output: Set of $\tilde{Z}$-matrices for each effect in $F$: $\zeta$
Function scaleInt(x, X, Y) is
    $y = x \times Y/X$;
    return $\lfloor y \rfloor$;
end
for $f$ in $F$ do
    $\zeta[f] = NaN(M \times N)$ ;                 /* Initialize ζ with Not-a-Number-matrices */
end
for $r$ in $R$ do
    $\tilde{Z} = \zeta[\text{faultEffect}(r)]$ ;   /* Get Z̃-matrix w.r.t.  the result's fault effect */
    $m = \text{scaleInt}(\text{injectCycle}(r), C, M)$ ;          /* Scale result's injection cycle */
    $n = \text{scaleInt}(\text{injectBit}(r), B, N)$ ;              /* Scale result's injection bit */
    $z_{m,n} = \tilde{Z}[m][n]$
    if $z_{m,n} = NaN$ then
        $z_{m,n} = 0$ ;                   /* Initialize if element was not accessed yet */
    end
    $z_{m,n} = z_{m,n} + 1$ ;                                   /* Increment element value */
end
return $\zeta$;
```

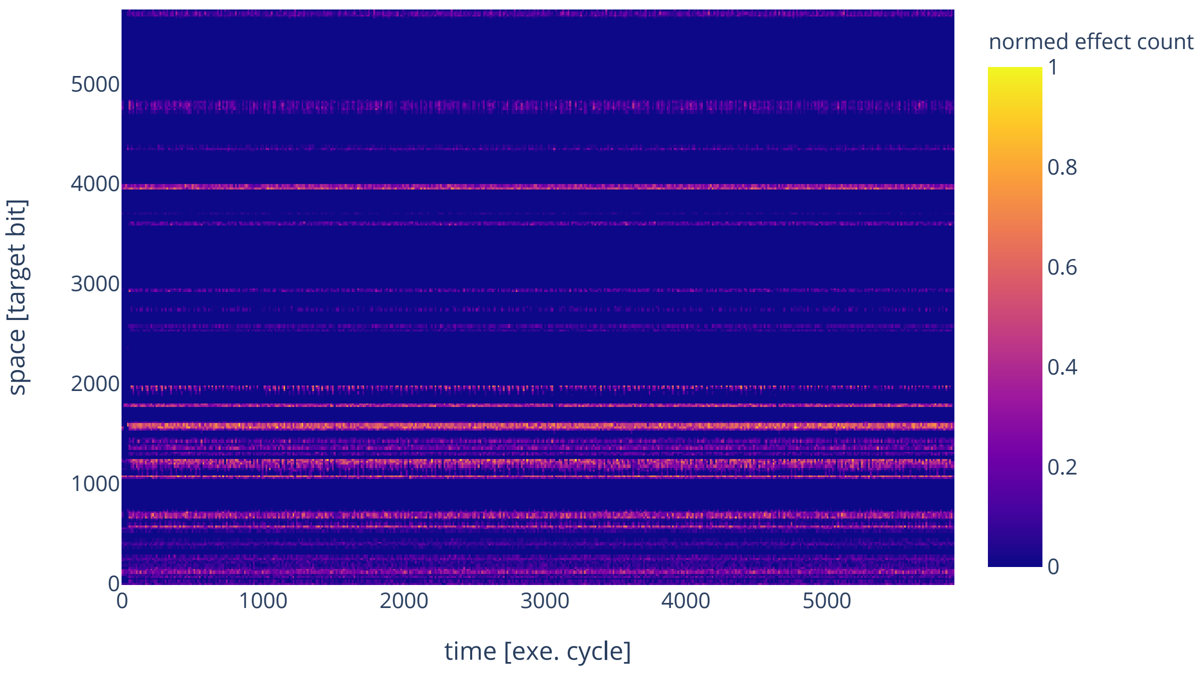


Figure 5.3.: Erroneous fault effects, ALU test - heat map

6. Results

6.1. Fault Effect Probability

After classification, a script counts the occurrences of a specific fault effect and sets these in proportion to the total number of simulation runs. The resulting statistic yields a probability figure for a random soft-error resulting in a certain fault. Additionally, the evaluation is further split into pure micro-architectural injections and injections into register file like parts of the model. All probability values have to be taken into consideration with respect to the confidence levels listed in tab. 5.2. First of all, the sum of all results is examined.

All Targets

group	count	%	key	count	%
Hang-Up Pipeline	4,374	0.0934	HP		
Hang-Up Logic	31,707	0.677	HL		
System Error	205,448	4.39	SM	123,294	60.0
			SI	82,154	40.0
Application Error	163,335	3.49	AO	102,519	62.8
			AP	106,478	65.2
Not Masked	2,317,138	49.5	NSK		
Masked	1,961,163	41.9	MSK		
$\sum$	4,683,165				

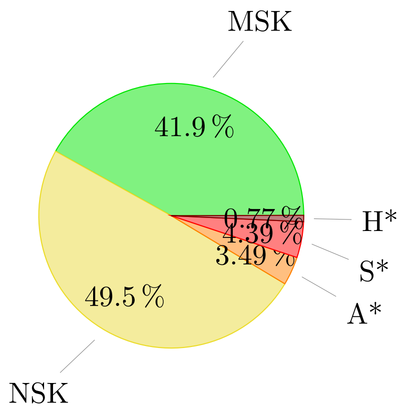


Table 6.1.: Fault effect probability, ALU test - all targets

Tab. 6.1 lists the ALU evaluation results for both micro-architectural and register file injections, i.e. all targets. Overall, 4374, or rather 0.0934 percent, of all injections lead to a hang-up where the pipeline gets stalled and 0.677 percent result in a hang where instructions are executed, but the application never finishes. Around 4.39 percent result in system faults of which about 40 percent are due to illegal instructions in the pipeline and 60 percent due to invalid memory access. These faults can be detected and handled during run-time. On the other side, silent corruptions in form of application errors make up about 3.49 percent of the simulations. Here, output memory is corrupted with a chance of 63 percent and program or execution flow with another 65 percent meaning an overlap of those two fault effects, i.e. the error corrupted both output and resulted in program execution deviations. 42 percent of all injections were entirely masked once a simulation run was completed, while about half of all the runs showed some deviation in the architectural states of the model but no error.

group	count	%	key	count	%
Hang-Up Pipeline	4,855	0.0975	HP		
Hang-Up Logic	39,082	0.785	HL		
System Error	206,399	4.14	SM	123,266	59.7
			SI	83,133	40.3
Application Error	186,538	3.75	AO	125,135	67.1
			AP	114,991	61.6
Not Masked	2,397,535	48.1	NSK		
Masked	2,145,771	43.1	MSK		
$\sum$	4,980,180				

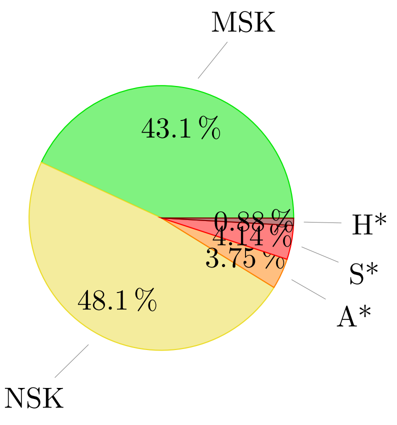


Table 6.2.: Fault effect probability, MUL test - all targets

group	count	%	key	count	%
Hang-Up Pipeline	2,305	0.0980	HP		
Hang-Up Logic	14,031	0.761	HL		
System Error	78,599	3.34	SM	39,786	50.6
			SI	38,813	49.4
Application Error	74,089	3.15	AO	47,350	63.9
			AP	69,355	93.6
Not Masked	1,150,042	48.9	NSK		
Masked	1,030,264	43.8	MSK		
$\sum$	2,353,196				

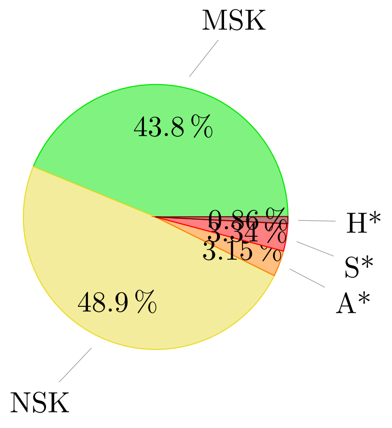


Table 6.3.: Fault effect probability, AES test - all targets

Comparing the different application or target software results with each other, there seems to be not much difference between the various application softwares with respect to fault effect probabilities. The AES test shows a smaller overall ratio of system and application level errors, however, almost all (94 %) of application level faults can be classified as program flow errors. The numbers for MUL and AES can be found in tab. 6.2 and 6.3.

Control and Status Registers

group	count	%	key	count	%
Hang-Up Pipeline	0	0	HP		
Hang-Up Logic	0	0	HL		
System Error	0	0	SM	0	0
			SI	0	0
Application Error	32,170	2.57	AO	0	0
			AP	32,170	100.0
Not Masked	1,069,434	85.4	NSK		
Masked	150,972	12.1	MSK		
∑	1,252,576				

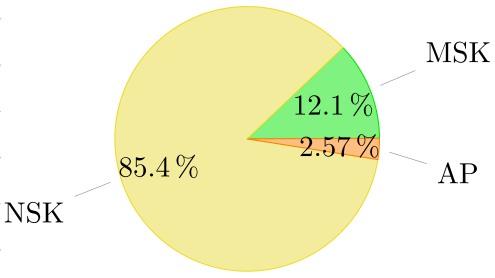


Table 6.4.: Fault effect probability, ALU test - CSR targets

Next, a look at injections into targets related to register file like hardware components is taken. This includes the components making up the GPR and CSR. Tab. 6.4 shows the ALU fault effect statistics for CSR target injections. In general, the statement can be made that these injections rarely result in hang-ups and system errors. However, program flow seems to be affected. The reason for this is, that the simulation framework uses the CSR to evaluate the number of executed instructions. With the classification from 5.1 that uses the returned instruction count from the framework, obviously, a deviation in execution flow occurs. This result has to be taken with a grain of salt, since only the instruction count storage might be affected and not the actual program flow. Manual inspection of these simulation runs confirm this. The instruction count is in a higher than possible number due to the maximum amount of clock cycles performed. After all, these errors can still be seen as such since application software might depend on performance and status registers. The results for the MUL and AES test are nearly equivalent to the ALU test. For completeness, the numbers can be found in tab. A.2 and A.5 in the appendix.

General Purpose Registers

group	count	%	key	count	%
Hang-Up Pipeline	0	0	HP		
Hang-Up Logic	28,664	3.56	HL		
System Error	42,867	5.32	SM	9,742	77.3
			SI	33,125	22.7
Application Error	21,703	2.69	AO	20,660	95.2
			AP	11,678	53.8
Not Masked	601,125	74.6	NSK		
Masked	111,842	13.9	MSK		
∑	806,201				

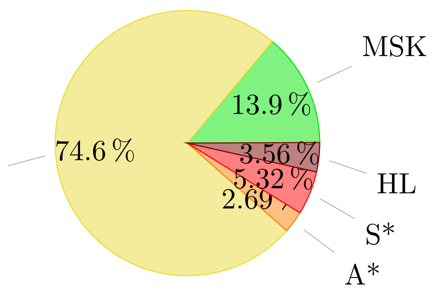


Table 6.5.: Fault effect probability, ALU test - GPR targets

Tab. 6.5 shows the fault effect statistics for GPR target injections. To interpret these GPR injection results a basic knowledge of the RISCV GPRs is helpful. In tab. 2.1 the GPRs and their fundamental usage is listed. The area making up the 32-bit hard-wired zero register (x0) is ignored, since the assumption was made that this component would be hard-wired, thus, not actually a sequential logic, i.e. flip flop, in real hardware. This leaves around 992 injectable bits or around one sixth of RI5CY's total injection targets. Hang-ups, however only logic stalls, occur with a probability of 3.56 percent. System errors with mostly illegal instructions being the cause occur with 5.32 percent. Application errors happen with 2.69 percent probability, while mostly affecting the model output. Interestingly, injections not affecting the program rarely result in a complete mask (13.9 percent), while with three quarters of all injections most remain in the model. The GPR as a component seems to be more vulnerable to hang-ups and system errors than the core as a whole. The large amount of illegal instruction system errors can be traced back to bit-flips to the GPR area where the stack pointer and function return address is stored (x1 and x2). Furthermore, the high occurrence rate of not-masked injections might be related to a large portion of the register file not being used, thus, not regularly updated during run-time by the ALU test program.

The results for the MUL and AES test are varying from the ALU test, meaning that fault effects resulting from injections into the GPR seem to be application dependent. The AES test might be more resilient to application level faults, which may be explained by the longer test masking stack-related memory errors more easily due to the higher access occurrences (the output is compared only at the end of each evaluation). The numbers for MUL and AES can be found in tab. A.1 and tab. A.4.

Micro-architectural Targets

group	count	%	key	count	%
Hang-Up Pipeline	4,374	0.168	HP		
Hang-Up Logic	3,043	0.117	HL		
System Error	162,581	6.26	SM	113,552	69.8
			SI	49,029	30.2
Application Error	109,363	4.21	AO	81,799	74.8
			AP	62,535	57.2
Not Masked	646,566	24.9	NSK		
Masked	1,672,240	64.6	MSK		
∑	2,598,167				

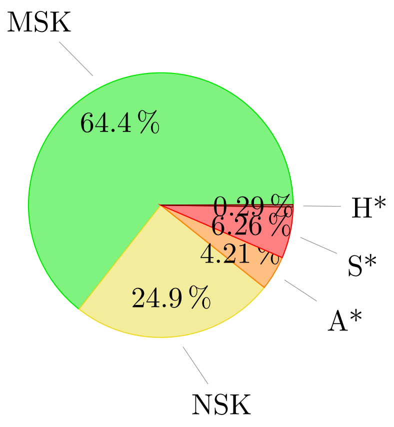


Table 6.6.: Fault effect probability, ALU test - micro-architectural targets

At last, micro-architectural-exclusive injections for the ALU software can be examined in tab 6.6. In contrast to GPR injections, micro-architectural injections show pipeline stalling hang-ups (0.172 percent), however, both, application and system level fault rates are higher with 4.2 and 6.3 percent respectively. While error free runs occur mostly at the same rate as they do with GPR injections, the probability of complete masking is with 66 percent higher in micro-architectural injections than in GPRs.

Compared to the ALU test, the AES result is not much different for micro-architectural injections. On the other hand, for the MUL test, a higher rate of application level faults can be observed. This might be explainable by the additional micro-architectural hardware (multiplier units) used in the MUL compared to ALU and AES test which solely utilize the ALU of the RI5CY core. In addition, AES has with 0.55 percent a much higher rate of logical hang-ups than its counterparts (0.117 for ALU and 0.119 for MUL). Again, the much longer execution time, thus, number of branching and jumps, resulting from the high repetitiveness of the crypto algorithm seem to increase the risk for logical hang-ups. The results can be found in tab. A.3 and A.6 for MUL and AES respectively.

6.2. Fault Effect Causation

The fault effect causes are examined according to the data processing ideas presented in sec. 5.2.2. Primarily, this survey focuses on causes for fault effects in the ALU test, however discrepancies between the ALU and the MUL or AES are taken into consideration. For better visibility, all causation cluster figures are scaled to the weighted fault effect proportion, while tables listing influential targets show both, weighted and unweighted values.

6.2.1. Hang-up Errors

The overwhelming portion of hang-ups are caused by injections in the GPR. In these cases the core's pipeline still executes instructions, however, the application does not finish because of some loop body might be infinitely executed due to a failing exit check. Such logic errors can be caught by watchdog-like system components checking the application's progress. More problematic are hang-ups caused by a stuck pipeline, i.e. no instructions propagate through the processing stages.

Hang-up - Pipeline Stall

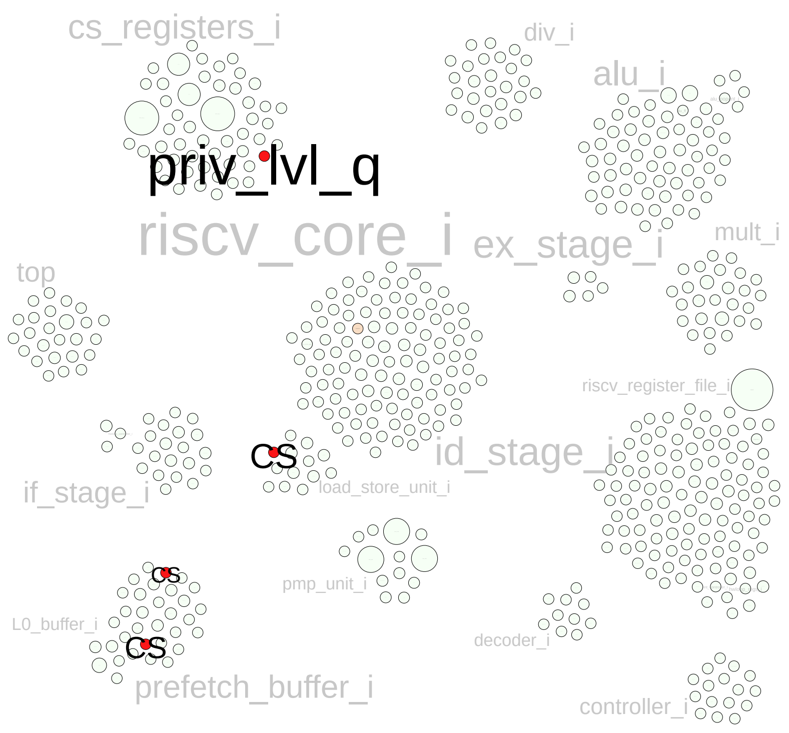


Figure 6.1.: Hang-up pipeline stall, ALU test - causation cluster

Fig. 6.1 shows the weighted causation cluster evaluation for pipeline stalling hang-up effects in the RI5CY core (ALU test). The intensity of (red) coloration and label size of each node or rather target indicate its proportional share in overall pipeline stalling hang-up faults. There is not much difference between the weighted and unweighted statistic which means that these hang-up errors mostly depend on the individual target, i.e. its function in the RISC-V core. With the exception of the MUL test, where the FSM in the utilized multiplier unit might cause additional pipeline stalls, the results are test independent.

target	**module**	**un- /weighted [%]**			**function**
		ALU	**MUL**	**AES**	
priv_lvl_q	cs_registers_i	37.0/39.0	36.2/37.8	35.1/36.8	RISC-V privilege level
CS	load_store_unit_i	23.1/24.4	21.9/22.8	23.7/24.8	current state FSM
CS	L0_buffer_i	22.2/21.3	22.2/21.5	21.9/20.9	current state FSM
CS	prefetch_buffer_i	17.5/15.1	17.1/15.3	15.7/13.4	current state FSM
mulh_CS	mult_i	0/0	1.74/1.69	0/0	current state FSM
csr_restore_uret_id	riscv_core_i	0.23/0.26	0.83/0.93	3.59/4.11	restore PC (user-mode handling)

Table 6.7.: Hang-up pipeline stall - influential targets

Tab. 6.7 lists the most influential injection targets with their respective causation ratio. These hang-ups are produced by injections into delicate registers, e.g. finite state machine (FSM) current state storage and the core's privilege level. For the FSM cases, the system does not seem to be able to regenerate by itself and, in some instances, gets stuck at fetching new instructions. Even worse is an injection in the privilege level register, since here an injection always leads to a stuck pipeline. Reason might be that the test program normally stays in machine mode and a flip changes either to supervised mode or sets the level to the reserved, or rather undefined, mode (see. tab. 2.2). The privilege level has crucial influence on the systems pipeline as it may result in failing instruction fetch attempts due to the memory protection unit blocking access.

Fig. A.21 shows the pipeline stall effect heat map for the ALU test. The lower line contains the FSM registers for the Prefetch Buffer and L0 Buffer. The upper line is built from data regarding the privilege level register and LSU FSM. Overall, both lines seem to be more or less continuous which indicates that these hang-up effects are not application/time dependent. However, further, more fine-grained, analysis of those targets' fault effect might reveal things like cycle or value-dependency. For example, injections in the current state register of these FSMs might only result in stalls if a certain state is active and then corrupted.

Hang-up - Logic Stall

Logic or application level stalls mostly result from injection into the GPRs memory elements (see tab. 6.8). Fig. 6.2 shows the causation cluster, while the heat map in fig. A.22 displays a more irregular pattern compared to pipeline stalling hang-ups. A closer inspection of the responsible simulations reveals that mostly the return address, stack and global pointer register regions are the causing sectors in the register file. In the heat map, some of the traces are discontinuous which suggests that their respective targets make the core vulnerable only at specific moments, thus, are influenced by the application code and application progress.

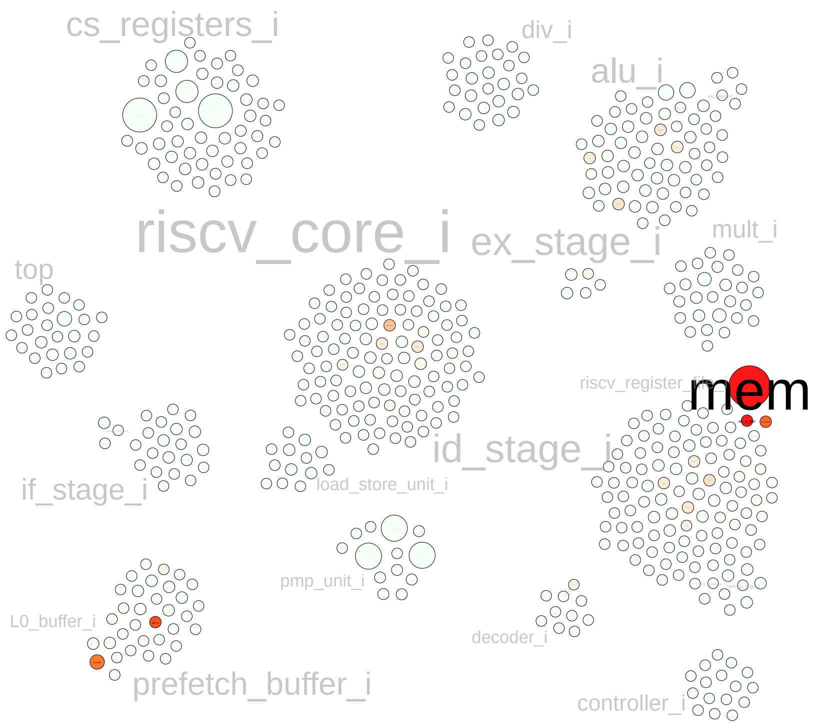


Figure 6.2.: Hang-up logic stall, ALU test - causation cluster

target	module	un- /weighted [%]			function
		ALU	MUL	AES	
mem	register_file_i	90.4/80.8	91.6/83.6	56.4/38.8	register file memory
we_a_dec	register_file_i	2.43/4.90	2.37/4.66	2.46/3.48	reg. file write enable port a
addr_q	prefetch_buffer_i	1.62/3.26	1.55/3.04	5.18/7.31	prefetch address
we_b_dec	register_file_i	1.35/2.72	1.60/3.16	2.07/2.92	reg. file write enable port a
L0_buffer	L0_buffer_i	1.17/2.22	0.54/1.01	6.87/9.21	128-bit instr. cache

Table 6.8.: Hang-up logic stall - influential targets

6.2.2. System Errors

Fig. 6.3 shows the ALU evaluation's system error causation cluster. Fig. A.10 focuses on memory access errors and fig A.9 illustrates illegal instruction fault causes. Again, the GPR register and components related to the instruction fetch have a high influence on this fault effect (see tab. 6.9). The corresponding heat map for the ALU test is depicted in fig. A.23.

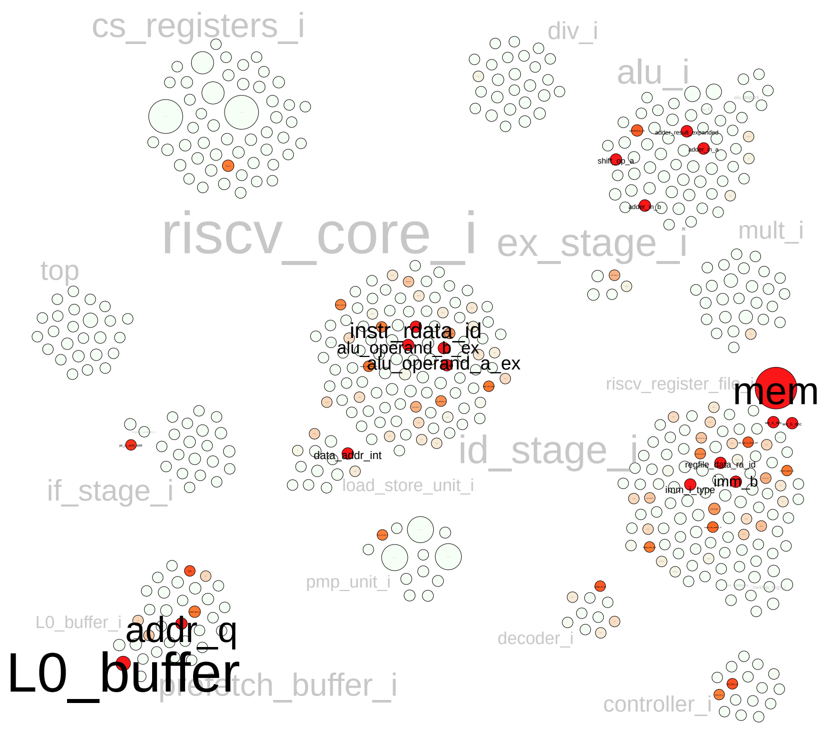


Figure 6.3.: System error, ALU test - causation cluster

System Error - Memory Access

Beside the LSU, IF, and ID pipeline components, ALU related targets as a part of the execution stage, show an additional high ratio of causation behavior for memory access violations by affecting potential load and store addresses. This can be related to the ALU being used to calculate memory addresses for storing the test values, thus, accessing invalid address spaces if the address value was corrupted.

target	module	un- /weighted [%]			function
		ALU	MUL	AES	
mem	register_file_i	20.9/11.7	16.1/8.64	23.7/13.5	register file memory
L0_buffer	L0_buffer_i	15.6/16.7	17.0/17.7	13.0/14.1	128-bit instr. cache
addr_q	prefetch_buffer_i	9.65/10.8	10.2/10.9	10.9/12.4	prefetch address
instr_rdata_id	riscv_core_i	6.47/6.27	6.27/6.88	5.38/6.14	instr. data (IF)
alu_operand_a_ex	riscv_core_i	5.47/5.14	5.14/5.65	4.57/5.20	ALU op. a (ID)
alu_operand_b_ex	riscv_core_i	5.05/4.75	4.74/5.21	4.38/4.99	ALU op. b (ID)
imm_b	id_stage_i	3.94/4.43	4.33/4.75	4.03/4.60	ALU immediate b
data_addr_int	load_store_unit_i	2.90/3.26	3.31/3.64	3.20/3.65	LSU addr. (EX)

Table 6.9.: System error causation - influential targets

System Error - Illegal Instruction

For illegal instructions mostly GPR or instruction fetch and decode related targets are an issue, while executing modules are not.

6.2.3. Application Errors

Fig. 6.4 shows the evaluation's application error causation cluster. Fig. A.15 focuses on targets silently corrupting the systems output and fig A.16 illustrates fault causes altering the application's execution progress. The overall fault pattern is very similar to the system error one. However, these application level errors are not as easily detectable by the core itself. For example, a storage address might have been corrupted but the altered address value might still be in a region viable to the memory protection. More classically, the actual data written to system outputs varies from what should have been calculated by the application. Overall, almost every module or region of the system can produce an application error. The heat map is shown in fig. A.24. Most prominently a thick and continuous line around bit index 4,000 can be seen which corresponds to the program counter registers.

Application Error - Output

For output corruptions, the most parts of the system under test can have an affect. The comparison between MUL and ALU/AES test shows, that even though some hardware parts are not utilized by an application, they can still produce fault effects. For example, in the ALU and AES tests, the multiplier and division units are not used, however, are identified by the framework as causing application level faults. In this specific case, these faults were caused, like in pipeline stalling effects, by current state registers of FSMs.

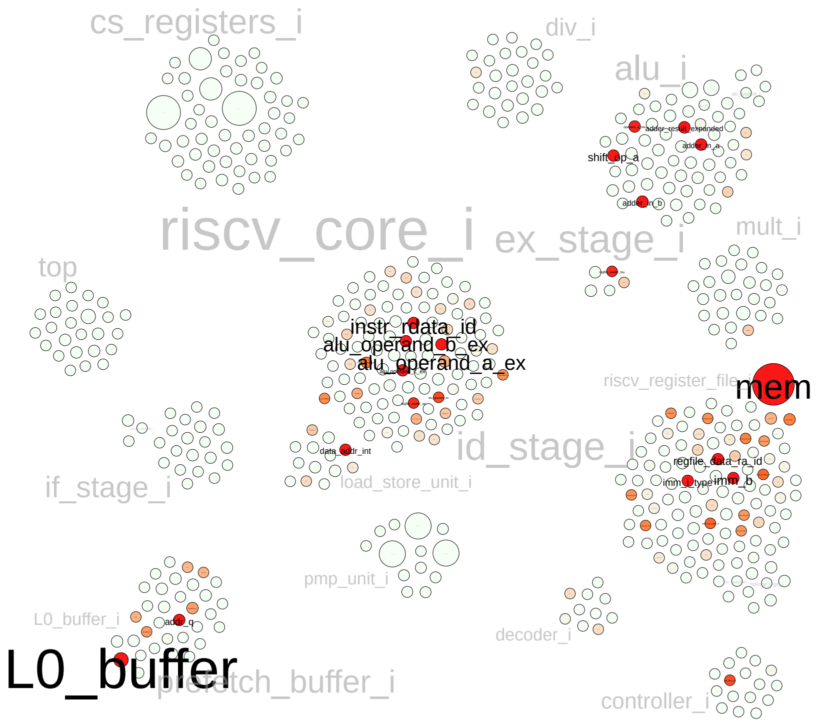


Figure 6.4.: Application error, ALU test - causation cluster

Application Error - Program Flow

For program or execution flow corruption, the pattern resembles the output corruption pattern, however, parts of the CSR take over as most influential ones, e.g. the program counter registers. This phenomenon is described in sec. 6.1.

target	**module**	**un- /weighted [%]**			**function**
		ALU	**MUL**	**AES**	
PCCR_q	cs_registers_i	15.9/15.0	14.9/14.0	17.1/15.7	performance counters
mem	register_file_i	13.3/9.27	10.7/7.55	5.12/3.46	register file memory
L0_buffer	L0_buffer_i	12.9/13.5	15.2/15.6	16.5/16.8	128-bit instr. cache
instr_rdata_id	riscv_core_i	4.57/4.94	4.34/4.63	5.67/5.967	instr. data (IF)
alu_operand_a_ex	riscv_core_i	3.98/4.30	3.60/3.84	3.98/4.18	ALU op. a (ID)
alu_operand_b_ex	riscv_core_i	3.85/4.16	3.21/3.42	3.73/3.92	ALU op. b (ID)

Table 6.10.: Application error causation - influential targets

6.3. Conclusion

Several consequences can be formulated for the evaluation results. The more statistical approach observing the probability of a fault effect appearing considering random injections can show an overall vulnerability of the evaluated RISC-V core to soft errors. In case of RI5CY with a bare-metal test software evaluated without any additional resiliency methods applied, the RISC-V core has shown a good response, i.e. soft error resiliency. The overall number of hang-up effect (pipeline and logic stalls), for example, might seem quite high at around 0.8 percent, however, the majority of these errors could be either handled or avoided by measures on the software level in the first place. For example, (Weinzierl 2017) explored multiple compiler-based solutions that check the control flow by software signatures or duplicate instructions to introduce redundancy. These software measures could reduce the rate of logic hang-ups observed in this thesis' evaluation.

Similarly, the system can handle the observed system errors itself provided its protecting components are configured in such a way and a defined state is reached by the system on its after a fault. However, the pipeline stalling faults would remain. This sub-group, which makes up about 0.1 percent of all evaluations results, has to be solved by hardware measures. In general, there are two approaches for this; Either resolving the fault effect or avoiding it in the first place.

Avoiding errors would require to increase the hardware's resiliency against the causing effects, i.e. neutron and alpha particle effects on the chip. While random access memory (RAM), both static and dynamic, are frequently considered for radiation hardening, micro-architectural parts often are not. (Baze et al. 2000) proposed radiation hardening techniques for CMOS logic through placement of transistors in biased and isolated wells. Amongst other things the work showed examples for resilient inverters that, in form of cross-coupled pairs, are the basic components for sequential logic. Electronic suppliers also provide radiation tolerant SoC platforms that, for example, can be found in aerospace applications (Microchip Technology Inc. n.d.). Resolving, or rather detecting and correcting an error, could be done by some additional hardware performing continuous checks. This could either be a quite forceful action like resetting the system to a error free state, e.g. by a watchdog, or a more gentle one; Sequential logic paths are implemented multiple times, then, a majority vote decides what the correct value for each path has to be since multiple soft-errors on the same path are very unlikely (Baumann 2005). This and other redundancy methods increase area usage of the protected hardware, therefore, are sometimes not feasible for the whole digital system. Implementing the redundancy only on critical sections could pose a trade-off between cost and the gained soft error resiliency.

The results from this work's evaluation process have shown some of these critical sections for the RI5CY core in form of vulnerable micro-architectural targets. Some form of majority vote could be added to the core's various finite state machine registers that were showing an especially high ratio of pipeline-stalling hang-up effects.

7. Summary

In this work, a fast, guaranteed fault injection framework based on the Verilator HDL simulator was developed. To enable Verilator's output (VRTL) simulations of soft errors, its C++ sources had to be modified. Due to the size of the VRTL, manual modifications were not feasible. Additionally, two modification types were evaluated with source-transformation coming out on top. Therefore, a LLVM-based and model-agnostic source transformation tool was developed that inserts injection points in the source code. An API builder automatically generates an individual Injection API for a each modified VRTL, thus, allowing high compiler optimization capabilities for executable simulation.

A possible design flow was proposed through a toolchain that builds an efficient injection framework from hardware description to executable simulation. As proof-of-concept, the modified VRTL of a RISC-V ISA implementing core was extended to a fault injection simulation framework. The framework was fed with over 10 million random injection runs for three different test program. In each simulation run, one space- and time-random bit flip of the model's sequential logic was applied and the effect on the core under test observed. From the mass of simulation run results, an evaluation for statistical probability of a certain fault effect was formulated. Furthermore, each fault effect was examined for causation in the model. Thus, existing vulnerabilities of certain parts of the core under test could be located.

Most fault effects can be handled either by additional features, like proper exception handling, of the examined core itself or by software-based resiliency enhancements. The remaining vulnerabilities, mostly pipeline stalling effects, would need further hardware-based measures.

In future works, the framework could be extended to support additional peripherals. Verilator's SystemC capabilities could be utilized to integrate high-level models of such hardware components, thus, enabling more complex virtual prototypes with fault injection capability. Furthermore, while the work for this thesis was carried out, Verilator added multi-threaded simulation support. Spreading the high CPU load for the RTL model to multiple cores of the host machine could speed up the simulation even further. The framework currently allows uncomplicated modeling of soft errors through random injections, however, deliberate faults could be simulated and serve as a basis for a fault attack evaluation of RISC-V cores.

A. Appendix

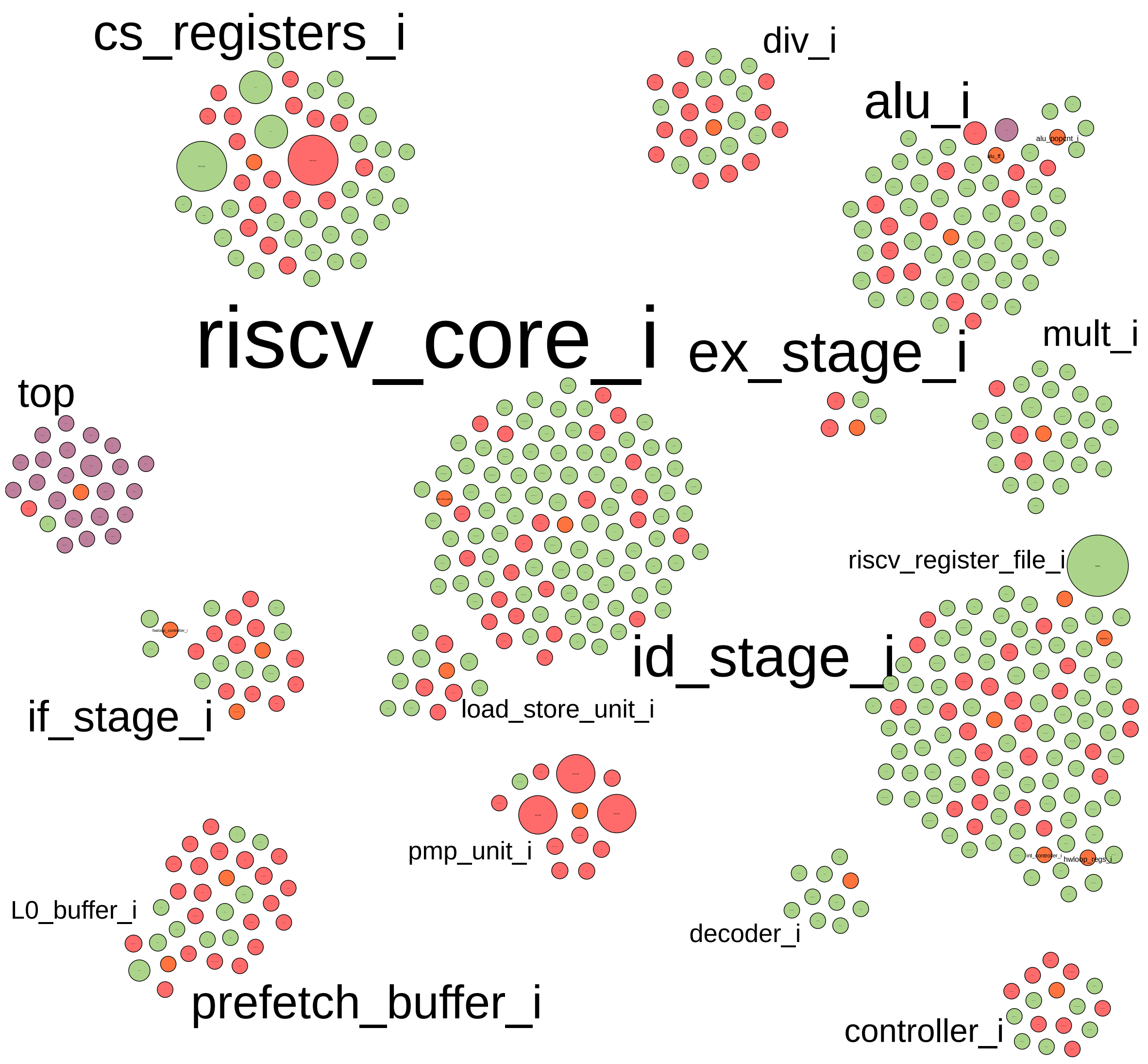


Figure A.1.: RI5CY injection target cluster

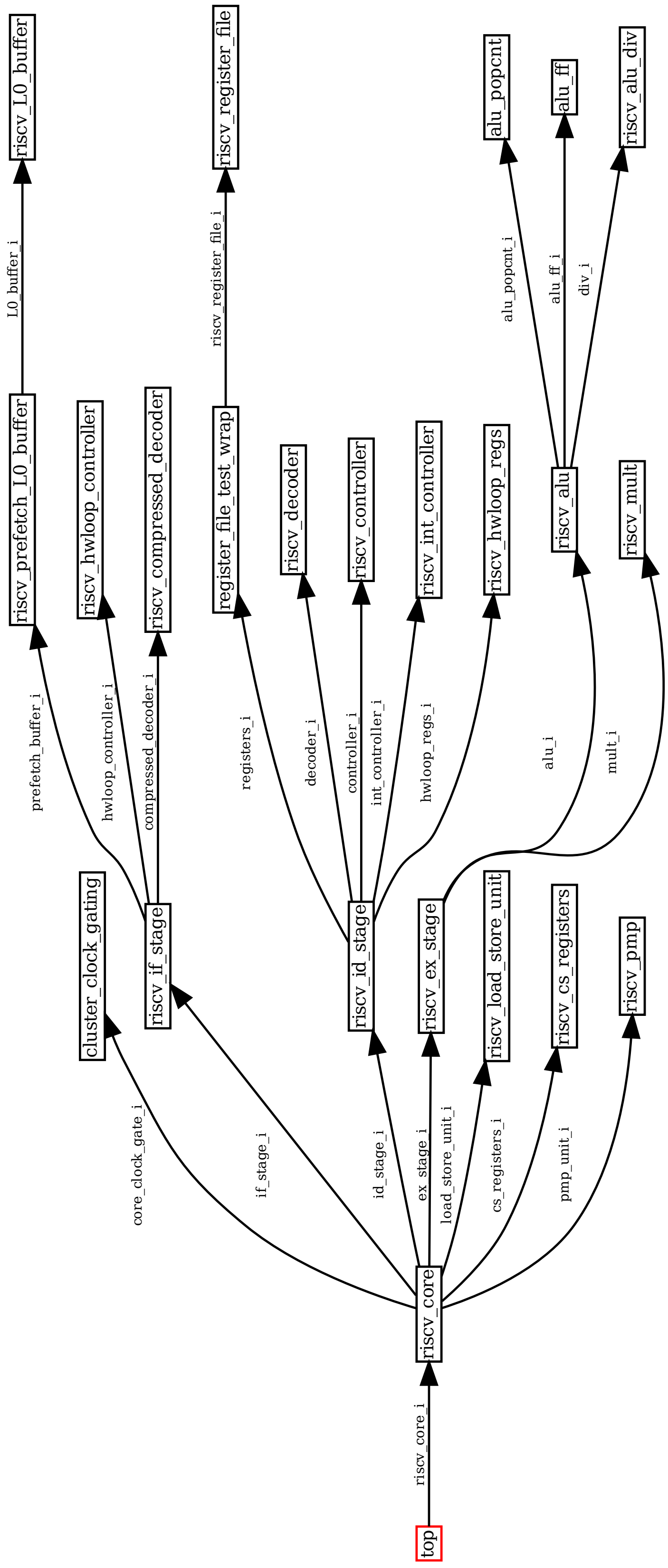


Figure A.2.: RI5CY hierarchy - modules and cell instances

Fault Effect Probability - MUL Test

group	count	%	key	count	%
Hang-Up Pipeline	0	0	HP		
Hang-Up Logic	26,796	4.19	HL		
System Error	24,766	3.87	SM	2,042	8.25
			SI	22,724	91.75
Application Error	14,609	2.28	AO	13,774	94.3
			AP	9,797	67.1
Not Masked	494,589	77.3	NSK		
Masked	79,096	12.4	MSK		
$\sum$	639,856				

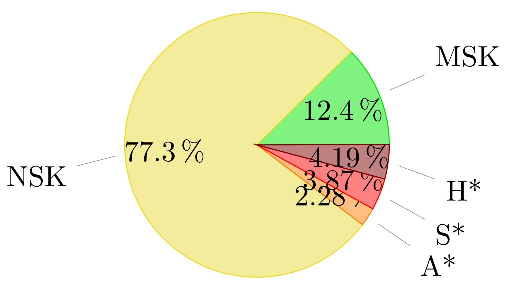


Table A.1.: Fault effect probability, MUL test - GPR targets

group	count	%	key	count	%
Hang-Up Pipeline	0	0	HP		
Hang-Up Logic	0	0	HL		
System Error	0	0	SM	0	0
			SI	0	0
Application Error	27,291	2.76	AO	0	0
			AP	27,291	100.0
Not Masked	845,367	85.4	NSK		
Masked	117,421	11.9	MSK		
$\sum$	990,079				

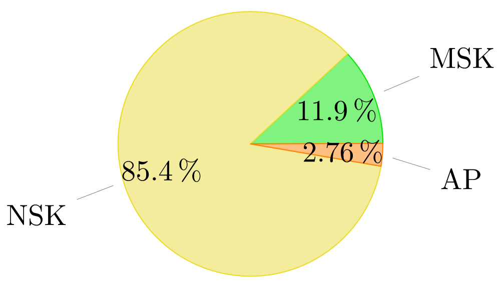


Table A.2.: Fault effect probability, MUL test - CSR targets

group	count	%	key	count	%
Hang-Up Pipeline	3,611	0.176	HP		
Hang-Up Logic	2,444	0.119	HL		
System Error	129,001	6.28	SM	90,189	69.9
			SI	38,812	30.1
Application Error	96,157	4.68	AO	78,889	82.0
			AP	47,903	49.8
Not Masked	444,918	21.7	NSK		
Masked	1,378,049	67.1	MSK		
$\sum$	2,054,180				

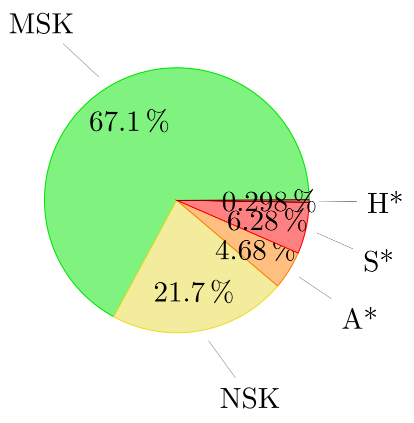


Table A.3.: Fault effect probability, MUL test - micro-architectural targets

Fault Effect Probability - AES Test

group	count	%	key	count	%
Hang-Up Pipeline	0	0	HP		
Hang-Up Logic	9,487	2.12	HL		
System Error	20,782	4.65	SM	7,143	34.4
			SI	13,639	65.6
Application Error	4,192	0.937	AO	3,914	93.4
			AP	3,813	91.0
Not Masked	317,342	70.9	NSK		
Masked	95,473	21.3	MSK		
∑	447,276				

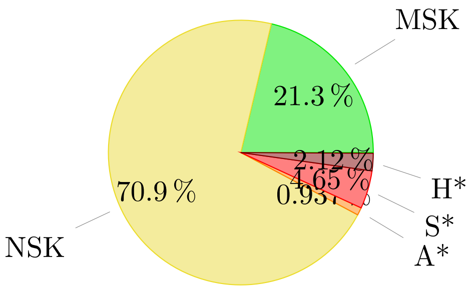


Table A.4.: Fault effect probability, AES test - GPR targets

group	count	%	key	count	%
Hang-Up Pipeline	0	0	HP		
Hang-Up Logic	0	0	HL		
System Error	0	0	SM	0	0
			SI	0	0
Application Error	18,407	2.66	AO	0	0
			AP	18,407	100.0
Not Masked	590,624	85.4	NSK		
Masked	82,903	12.0	MSK		
∑	691,934				

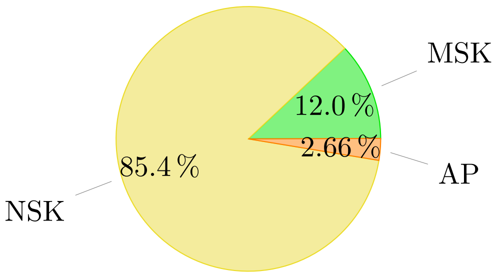


Table A.5.: Fault effect probability, AES test - CSR targets

group	count	%	key	count	%
Hang-Up Pipeline	2,504	0.174	HP		
Hang-Up Logic	7,368	0.513	HL		
System Error	66,927	4.66	SM	37,110	55.5
			SI	29,817	44.5
Application Error	61,044	4.25	AO	49,454	81.0
			AP	56,258	92.2
Not Masked	357,192	24.9	NSK		
Masked	941,600	65.5	MSK		
∑	1,436,635				

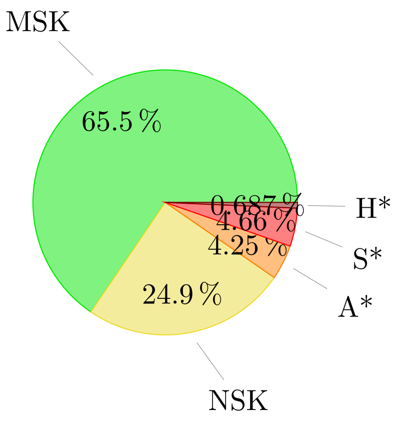


Table A.6.: Fault effect probability, AES test - micro-architectural targets

Fault Effect Causation - Hang-up Error Clusters

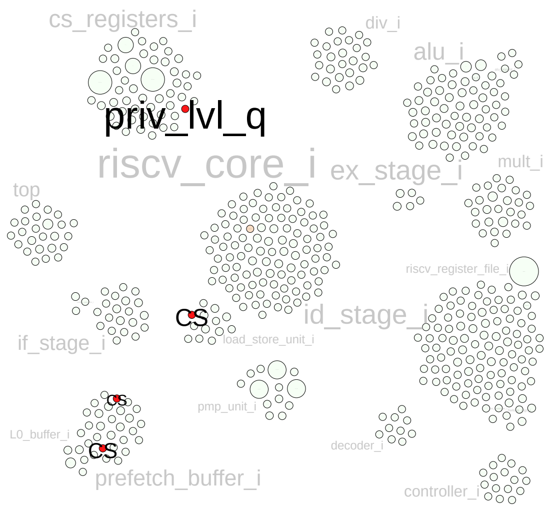


Figure A.3.: (HP) Pipeline stall, ALU test

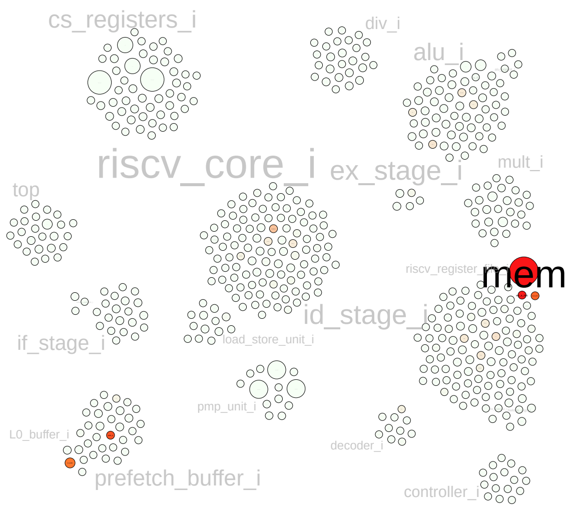


Figure A.4.: (HL) Logic stall, ALU test

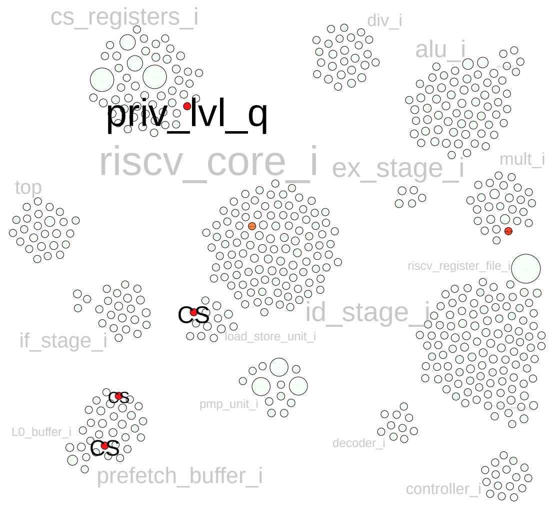


Figure A.5.: (HP) Pipeline stall, MUL test

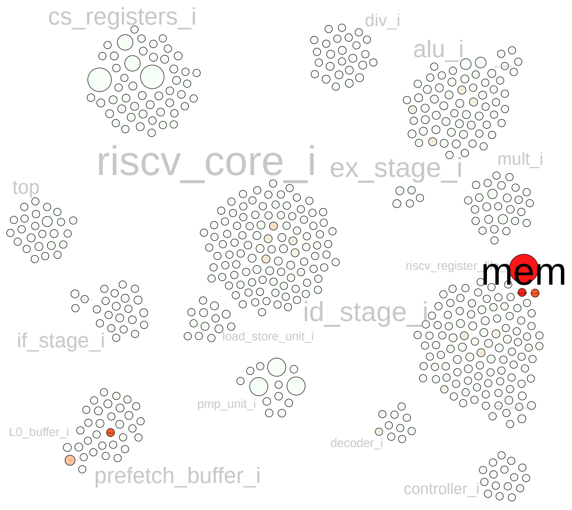


Figure A.6.: (HL) Logic stall, MUL test

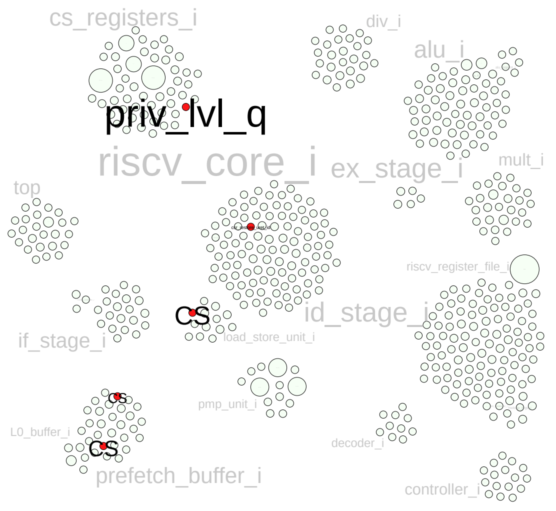


Figure A.7.: (HP) Pipeline stall, AES test

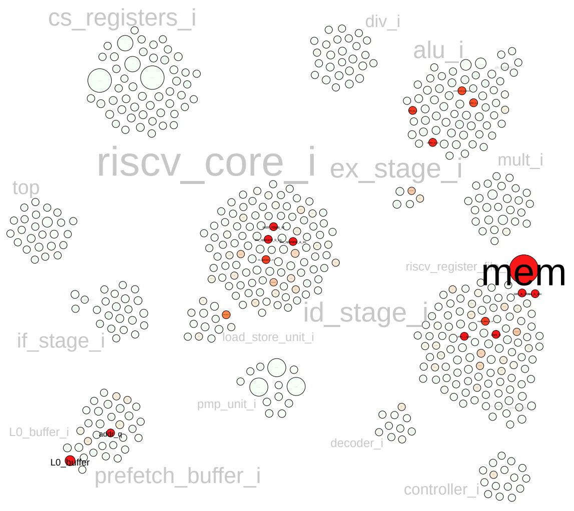


Figure A.8.: (HL) Logic stall, AES test

Fault Effect Causation - System Error Clusters

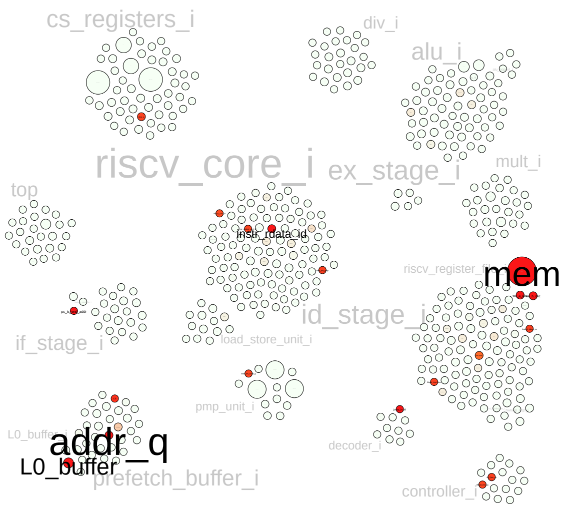


Figure A.9.: (SI) Illegal instruction, ALU test

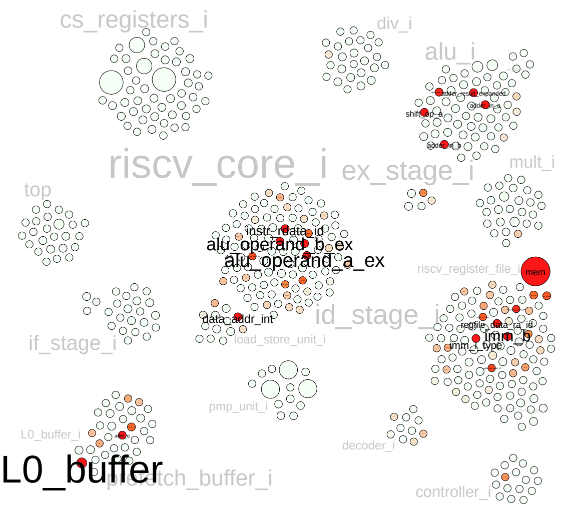


Figure A.10.: (SM) Memory access, ALU test

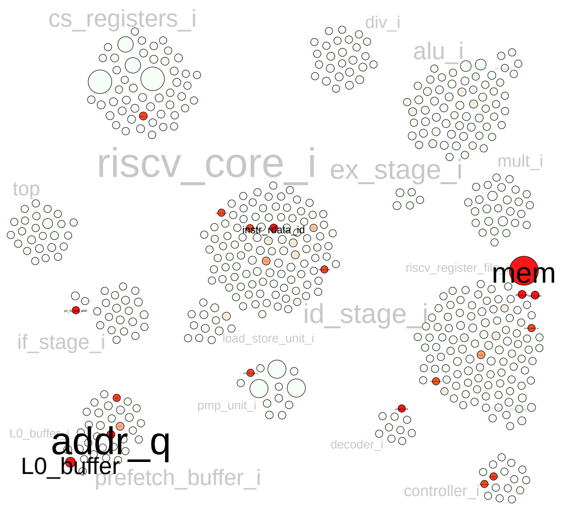


Figure A.11.: (SI) Illegal instruction, MUL test

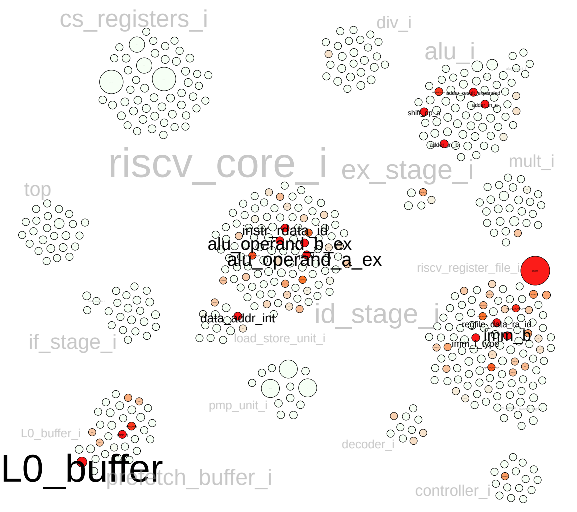


Figure A.12.: (SM) Memory access, MUL test

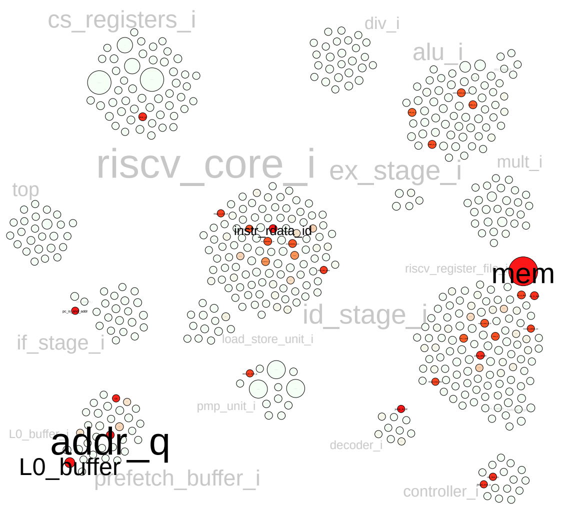


Figure A.13.: (SI) Illegal instruction, AES test

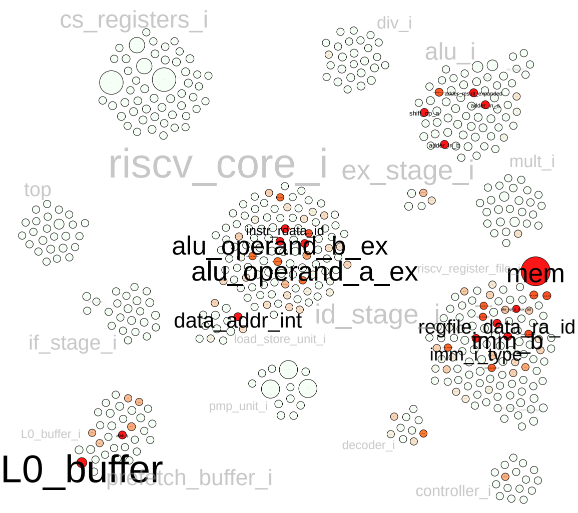


Figure A.14.: (SM) Memory access, AES test

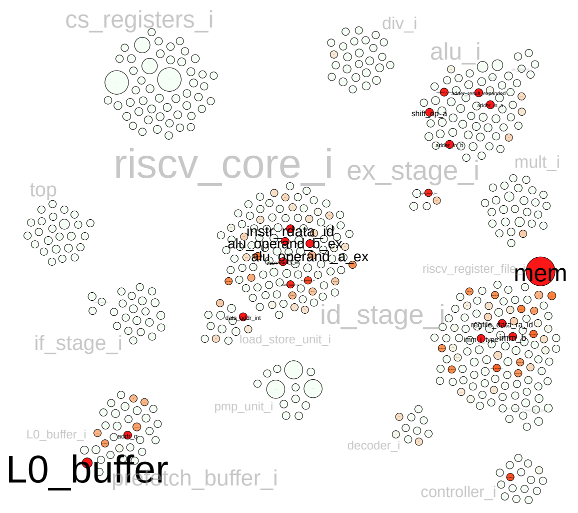


Figure A.15.: (AO) Output, ALU test

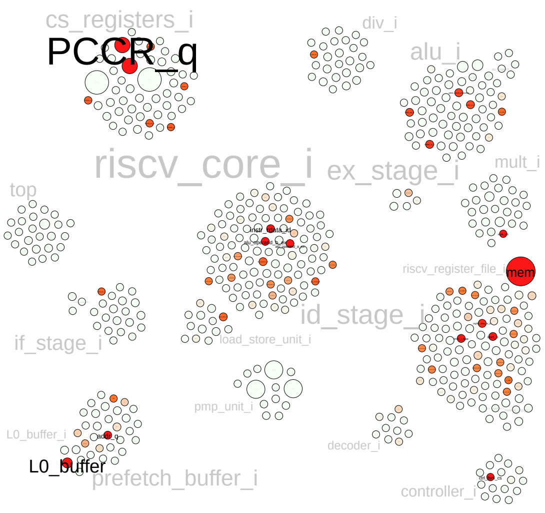


Figure A.16.: (AP) Program flow, ALU test

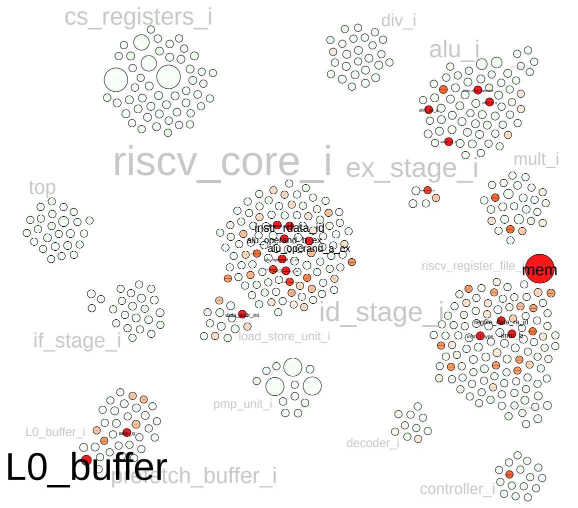


Figure A.17.: (AO) Output, MUL test

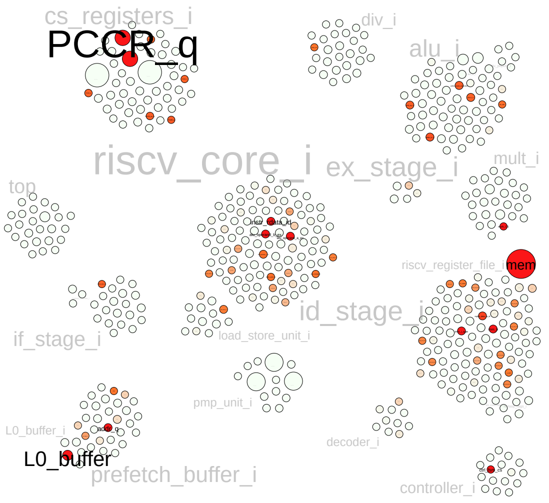


Figure A.18.: (AP) Program Flow, MUL test

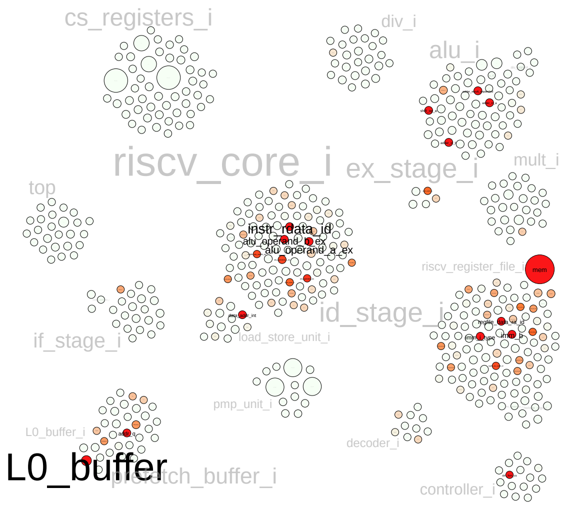


Figure A.19.: (AO) Output, AES test

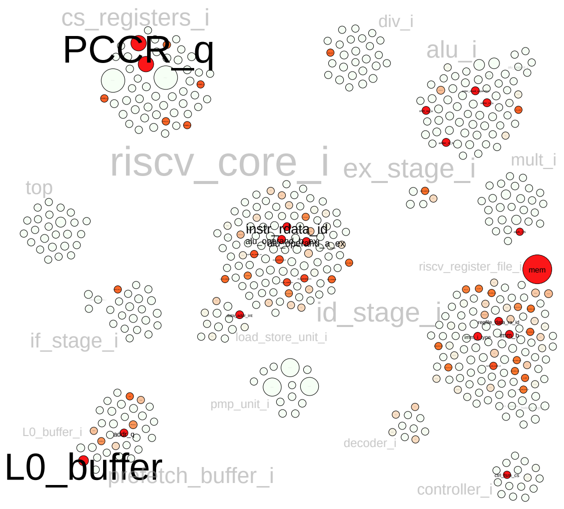


Figure A.20.: (AP) Program flow, AES test

Fault Effect Causation - Heat Maps

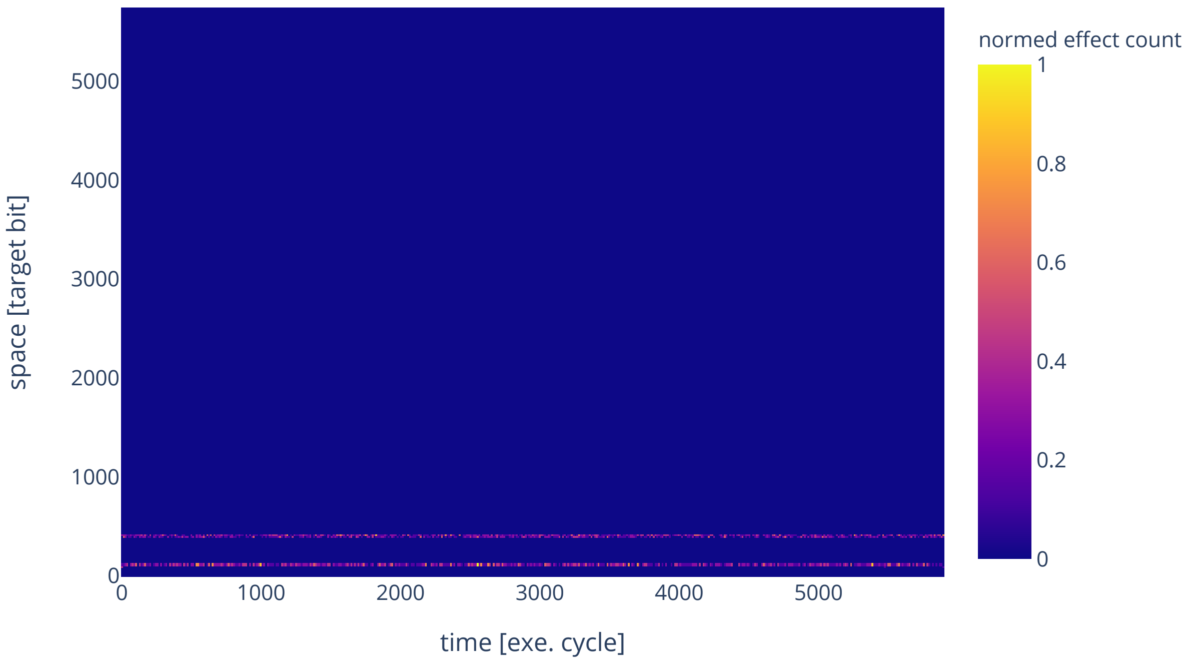


Figure A.21.: Hang-up pipeline stall, ALU test - heat map

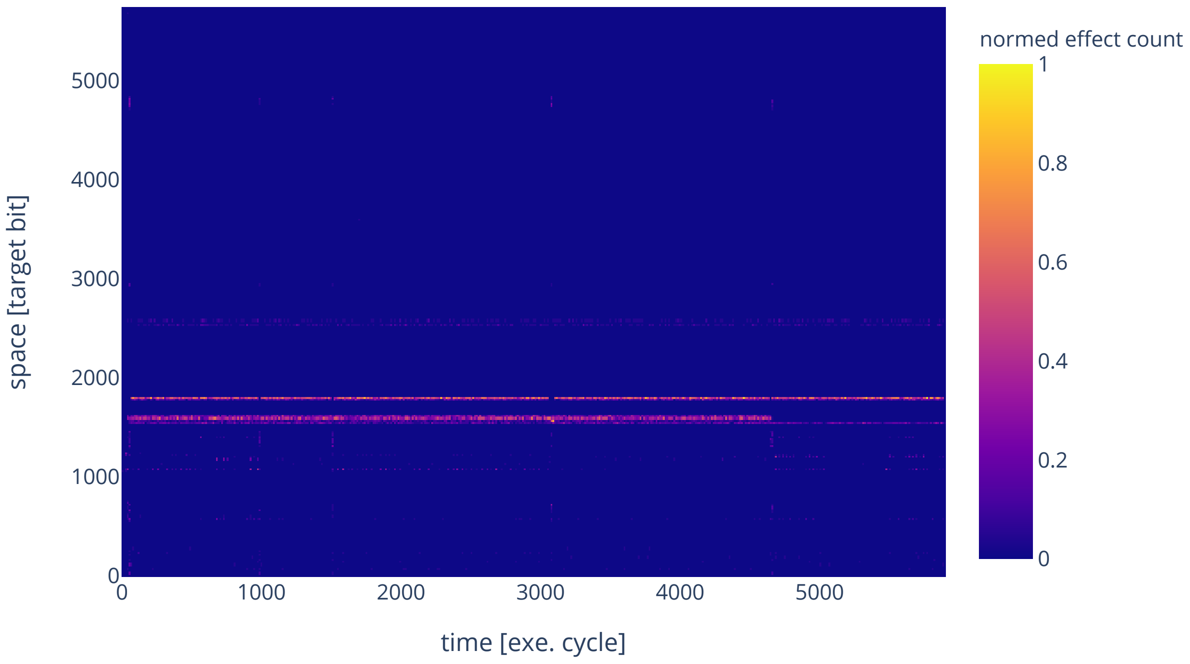


Figure A.22.: Hang-up logic stall, ALU test - heat map

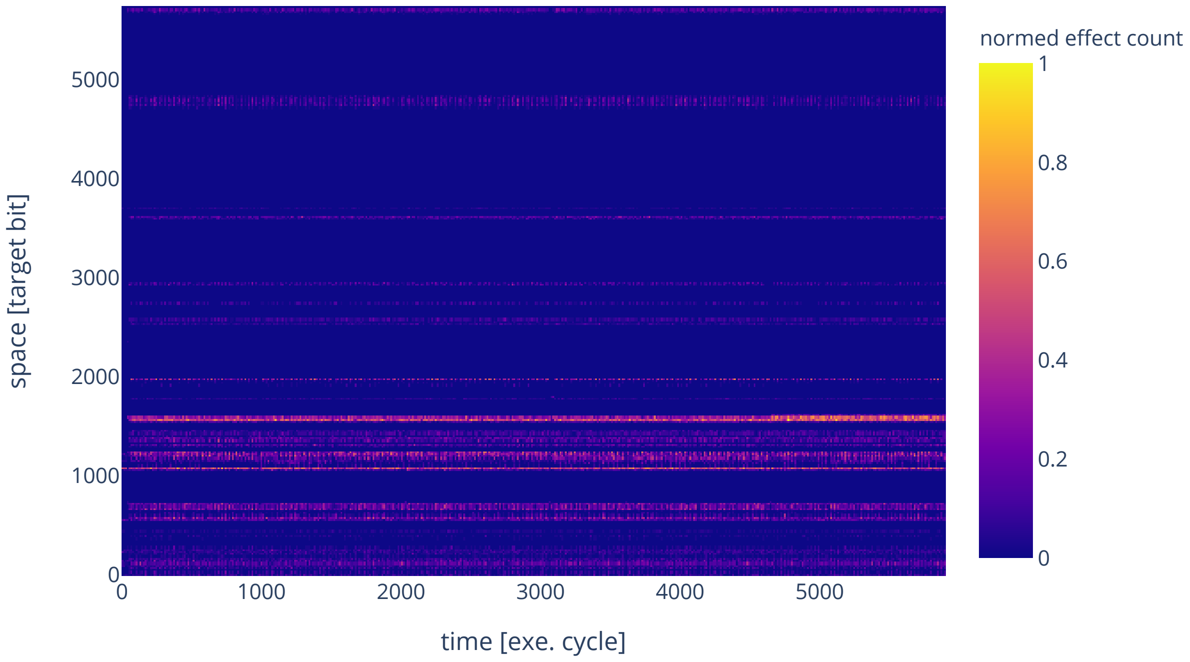


Figure A.23.: System errors, ALU test - heat map

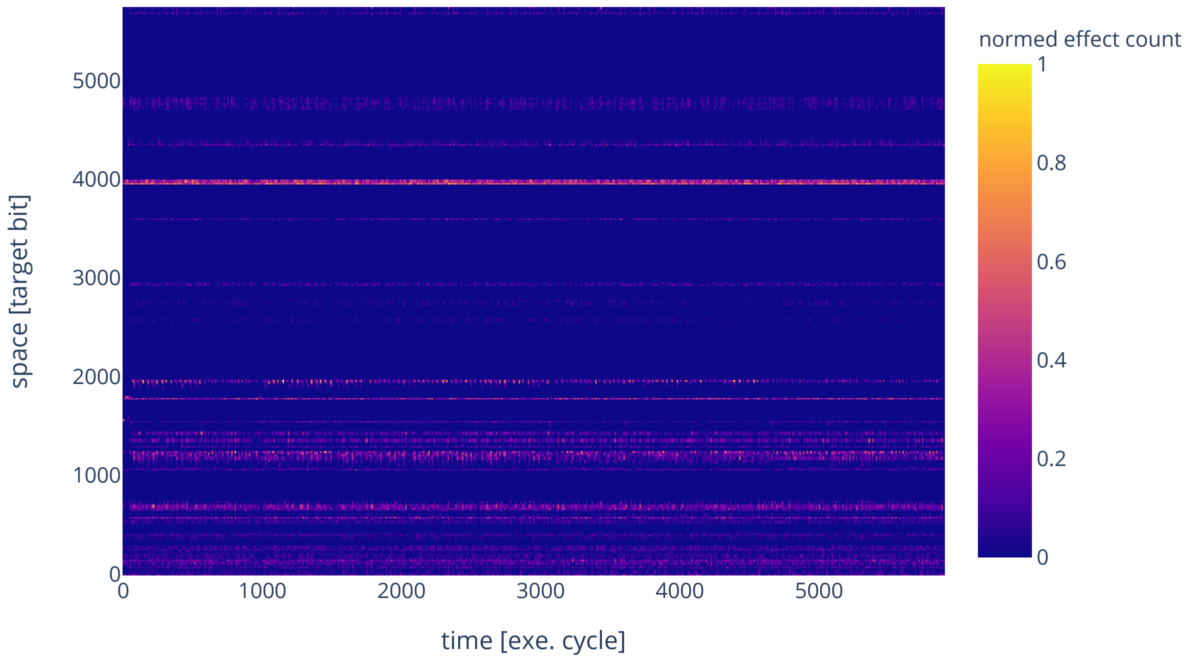


Figure A.24.: Application errors, ALU test - heat map

Bibliography

Baumann, R. (2005): Soft errors in advanced computer systems, IEEE Design Test of Computers **22**(3): 258–266.

Baze, M. P., Buchner, S. P. & McMorrow, D. (2000): A digital cmos design technique for seu hardening, IEEE Transactions on Nuclear Science **47**(6): 2603–2608.

Cho, Hyungmin, Mirkhani, Shahrzad, Cher, Chen-Yong, Abraham, Jacob A. & Mitra, Subhasish (2013): Quantitative evaluation of soft error injection techniques for robust system design, 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC) S. 1.

Dittrich, Martin (2017): Schnelle Fehlereffektsimulation für Soft-Errors in der Pipeline von RISC-Prozessoren, Master's thesis, TUM.

ETH Zürich, University of Bologna (n.d.a): PULP platform.
`https://pulp-platform.org/`

ETH Zürich, University of Bologna (n.d.b): RI5C-Y: RISC-V core.
`https://github.com/pulp-platform/riscv`

Geier, Johannes (2019): Automated extraction and categorization of signals in register transfer level simulators, Student report, TUM.

Gephi (n.d.): Gephi force atlas 2.
`https://github.com/gephi/gephi/wiki/Force-Atlas-2`

IEEE Computer Society (2001): IEEE Standard Verilog hardware description language, Bd. 2005.

IEEE Computer Society (2009): IEEE Standard for SystemVerilog - Unified hardware design, specification, and verification language, IEEE Standards Association **2012**(November): 1285.

Krste Asanovic, Rimas Avizienis, Jacob Bachmeyer, Christopher F. Batten, Allen, J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, David Chisnall, Paul, Clayton, Palmer Dabbelt, Stefan Freudenberger, Jan Gray, Michael Hamburg, John Hauser, David, Horner, Olof Johansson, Ben Keller, Yunsup Lee, Joseph Myers, Rishiyur Nikhil, Stefan O'Rear, Albert Ou, John Ousterhout, David Patterson, Colin Schmidt, Michael Taylor, Wesley Terpstra, Matt Thomas, Tommy Thorn, Ray VanDeWalker, Megan Wachs, Andrew Waterman, Robert Wat, Son & Zandijk., Reinoud (2017): The RISC-V Instruction Set Manual v2.2, 2012 IEEE International Conference on Industrial Technology, ICIT 2012, Proceedings **I**: 1–32.

Leveugle, R., Calvez, A., Maistri, P. & Vanhauwaert, P. (2009): Statistical fault injection: Quantified error and confidence, 2009 Design, Automation Test in Europe Conference Exhibition, S. 502–506.

LLVM Foundation (n.d.): The LLVM compiler infrastructure.
`https://llvm.org/`

Maniatakos, Michail, Karimi, Naghmeh, Tirumurti, Chandra, Jas, Abhijit & Makris, Yiorgos (2011): Instruction-level impact analysis of low-level faults in a modern microprocessor controller, IEEE Transactions on Computers **60**(9): 1260–1273.

Microchip Technology Inc. (n.d.): Radiation tolerant devices.
`https://www.microchip.com/design-centers/aerospace-and-defense/radiation-tolerant`

RISC-V (n.d.): RISC-V GNU toolchain.
`https://github.com/riscv/riscv-gnu-toolchain`

SiFive (n.d.): RISC-V QEMU part 1: Privileged ISA v1.10, HiFive1 and VirtIO.
`https://www.sifive.com/blog/risc-v-qemu-part-1-privileged-isa-hifive1-virtio`

Traber, Andreas, Gautschi, Michael & Schiavone, Pasquale Davide (2019): RI5CY : User Manual, (August): 1–57.
`https://www.pulp-platform.org/docs/ri5cy{_}user{_}manual.pdf`

TUM EI EDA (n.d.): ETISS repository.
`https://github.com/tum-ei-eda/etiss`

Veripool (n.d.): Introduction to Verilator.
`https://www.veripool.org/wiki/verilator`

Waterman, Andrew, Lee, Yunsup & Patterson, David (2014): The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, **I**.

Weinzierl, Josef (2017): Cross-layer resilience against soft errors for embedded control systems, Master's thesis, TUM.