Automated fault injection in Verilog hardware designs

Raven Szewczyk

4th Year Project Report
Computer Science and Physics
School of Informatics
University of Edinburgh

Academic year 2019/2020

Abstract

Making hardware more resilient to radiation effects is important for many space and terrestrial applications. In this project a tool is created for automatically injecting faults into Verilog hardware designs, working in simulations and in FPGA hardware implementations. This helps in identifying potential areas for fault tolerance improvements and debugging fault-tolerant designs in presence of bit flips and latch-up/downs. Verinject is more flexible than previously developed tools, as it is not tied to a particular vendor of hardware simulation or synthesis toolchain.

The tool – Verinject – has been tested using a variety of methods, including unit tests and comparing experimental results with a mathematical analysis for a simple circuit. It is shown that this tool correctly reproduces the predicted fault rates and behaviours in the analysed circuit. The cost of fault injection testing is shown to be insignificant performance-wise, and limited by a factor of 2 in terms of hardware cost compared to an equivalent testbench without injection. A testbench running on a TUL Pynq-Z2 FPGA board was 38x faster than a comparable testbench simulated with the open-source tool Icarus Verilog. The tool also worked correctly on larger designs: an Intel 8051-compatible core and a RISC-V processor used in labs for the CArD course.

Contents

Declaration

I declare that this report was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Raven Szewczyk

Chapter 1

Introduction

1.1 Motivation

With more and more aspects of our lives depending on technology, it's important to ensure that the hardware and software used is reliable and safe. As computing integrated circuits get smaller and use lower voltages, they can become more prone to errors due to charged particles from cosmic rays. This is mainly a concern in space environments, as on Earth its magnetic field shields electronics from the majority of charged cosmic rays. However, it is also relevant for safety critical terrestrial systems, where the probability of failure of a system must be minimised. One example effect of radiation impacting a terrestrial computer could be seen in a local election in Schaerbeek, Belgium – "an Single Event Upset gave a candidate an extra 4096 votes" as reported by The Independent [Joh17].

Motivated by the importance of radiation hardening of electronics, this project aims to aid verification of hardware designs under the influence of single event effects. This is enabled by the widespread use of Hardware Description Languages, such as (System)Verilog. They provide a standardised representation of hardware designs that can be directly synthesized into FPGA bitstreams or physical integrated circuit descriptions used in manufacturing, while remaining relatively high-level. This way, the source designs can be automatically transformed to introduce additional checks in a manner that is much less prone to error than fully manual testbench creation, especially for large designs, and it is not tied to any single ASIC/FPGA vendor's toolset either.

1.2 Objectives

The primary goal of this project is to create and evaluate a tool for automatically verifying Verilog hardware designs for reliability in presence of single event upsets. The sub-goals forming the primary goal are as follows:

- Write a (System)Verilog source code transformation tool that adds injection points

to the reference designs

- Set up a test harness for injecting faults and measuring their impact on the outputs
- Test the tool on small designs, for which theoretical fault rates can be easily proven
- Test the tool on at least one large design and compare results with literature
- Benchmark the logic complexity (area) and time cost of verification with fault injection compared to verification without injection

1.3 My contributions – achieved in this project

- Creating a (System)Verilog source code transformation tool – Verinject – used for adding injection points:
  - Making a formatting-preserving Verilog parser
  - Parsing Verilator's (open source tool) XML AST output to aid transformation
  - Making the fault injection transformation
- Creating an memory-efficient method of fault injection into Verilog designs without tying the solution to a particular simulation framework
- Designing and implementing an algorithm for uniquely addressing all memory components in a Verilog design based purely on the source code
- Designing and implementing supporting Verilog modules handling the fault injection process
- Making scripts for generating reproducible and controllable inputs for the fault injector
- Designing and implementing a memory-mapped FPGA interface for the fault injector, based on an open-source AXI bus slave example
- Testing the tool on various small Verilog code samples
- Designing a simple "array adder" circuit and predicting its behaviour under fault injection with the use of mathematical techniques
- Comparing the predictions with the results from running Verinject on the circuit, in simulation and on an FPGA
- Evaluating the rough cost (in terms of performance and hardware) of fault injection based on the experiment
- Generating correct Verilog with fault injection for an Intel 8051-compatible design

- Generating correct Verilog with fault injection for a RISC-V code used in the Computer Architecture and Design course
- Measuring the dynamic cross-section of a matrix multiplication program running on the RISC-V processor, and comparing results with literature

The most important contribution of this project is efficient fault injection into HDL-based hardware designs without tying it to a particular synthesis or simulation tool, like in most of the works evaluated in the following section. This method is also more efficient for hardware using large memory arrays, which was a problem for multiple other fault injectors. This was achieved by careful algorithm and data structure design, by making sure they work with just source-level structures in Verilog, but take into account their cost in real hardware. This approach was more difficult than other fault injection projects, but it led to a more flexible tool in the end.

1.4 Related work

The concept of verifying hardware designs in presence of externally induced faults is not new. Therefore to evaluate what novelty this project brings, existing work is reviewed below.

The most accurate way to measure the influence of radiation on hardware designs is to physically bombard them with ions and measure the results, as done by Evans et al. [Eva+17]. This method requires the target design to be placed under a targetable ion beam. Such equipment is expensive and the tests are more difficult to perform than software or FPGA-based simulation of faults, so this method is not suitable for cheaper designs or the prototyping phases of other designs. However, this study provides reference data for how effective the software methods are.

The authors have shown that “when SEUs were injected, 33% of the time it was possible to obtain a simulation result that was exactly identical to the fault produced by the targeted ion”. On top of that, 51% reproduced a fault, but did not produce an identical trace, which means about 84% of faults can be reproduced by simulating single event upsets. By also simulating single event transients in combinatorial logic, the simulations can reproduce 87% of the experimental faults, which is not a very large improvement, and warrants simulating only SEUs, as this project does. This study used a physical accelerator and scripts for the proprietary ModelSim simulator that injected faults at gate-level providing very accurate results, while this project tries to provide a more affordable and generic solution not tied to one piece of simulation software.

Simbah-FI [SBB19] is a hybrid fault injector, implemented in the Tcl scripting language for

the ModelSim simulator. It can simulate both transient and permanent faults, and injects them at gate-level. However, before and after fault injection the circuit is simulated on a behavioural level, which leads to a faster simulation. The problem with this approach is that the simulation is tied to ModelSim as is the previous approach, and can't be used on an FPGA to accelerate the testing process. The paper also makes reference to multiple other related works in its Section 2, most of which work on a high level – like modifying CPU instructions – and others are not automatic, so they're not as useful for comparison with this project.

NETFI [MV13] takes a completely different approach, it modifies the basic cell library of Xilinx tools. It achieves this by taking the output netlist of Synplify Pro, a synthesis tool, and adds extra logic at every Flip-Flop, RAM Block and combinatorial logic components. The results for a 6x6 matrix multiplication program under injection running on a 8051-compatible core led to a global error rate of 47.29%, agreeing with radiation ground testing presented in [Rez01]. It also adds a SRAM memory controller to store the results from the device tested. This approach allows NETFI to perform fault injection into proprietary IP cores, because it works at a netlist level, and it's independent of the HDL used – Verilog or VHDL. However, it targets the toolchain of a single FPGA manufacturer.

FITO [SM08] is much closer in many respects to this project, than the works evaluated above. It also works on Verilog source-level, introducing transient (in `wires`) and single event (in flip-flops) faults. Because it also produces synthesizable results, the authors were able to measure an 80x increase in speed of testing when experiments were ran on and FPGA compared to simulations. However, the main limitation of this injection tool is that it doesn't handle memory cells, instead it decomposes them into individual flip-flops, which can be very costly for designs using RAMs and ROMs extensively. It can also only model stuck-at faults at wire boundaries, and not memory cells. Verinject does not attempt to introduce transient faults into logic, however it can handle SELs in memory cells.

A summarized comparison between the injectors described above is provided in the following table:

Name	**Point of injection**	**Dependencies**	**SEUs**	**SELs**	**SETs**
Evans et al. [Eva+17]	Physical chip	Ion accelerator	✓	✓	✓
Evans et al. [Eva+17]	Gate simulation	ModelSim	✓	✓	✓
Simbah-FI [SBB19]	Gate simulation	ModelSim	✓	✓	✓
NETFI [MV13]	Netlist	Synplify&Xilinx	✓	×	✓
FITO [SM08]	Verilog	Minimal	✓	✓/×	✓/×
Verinject (this project)	Verilog	Minimal	✓	✓	×

1.5 Results overview

During this project, a a working fault injection tool – Verinject – was created. It has been demonstrated on small Verilog code examples as well as slightly larger designs that it can handle a lot of the syntactic complexity of the Verilog language. Moreover, the fault injection process itself has been validated by analysing its results on an example that's simple enough to be mathematically tractable.

An experiment with an array addition circuit, producing sums of a set of input elements, led to the following conclusions:

- The fault injection used by Verinject yields to behaviour agreeing with mathematical theory
- For this circuit, the probability of a fault in the input affecting the output is slightly above one half
- The distribution of how many bits in the output change in response to a single bit in the input changing follows a geometric series with a ratio of $\frac{1}{2}$
- Verinject generates the same test results for the same configuration parameters, confirmed for 1000 cases in simulation and on an FPGA
- FPGA testing was 38x faster than simulation for this simple example
- Adding fault injection slowed down the resulting synthesized FPGA circuit by 1.6%
- Adding fault injection required slightly less than double logic resources on the FPGA, with much lower cost for large memory modules

Running fault injection on a RISC-V processor was successful, and for a 6x6 integer matrix multiplication program led to a 23.1% probability that the result was affected. Compared with ground radiation testing of a much simpler 8051 processor [Rez01], which led to a 46.71% result, it is reasonable considering the program was only using about half of the 32 available registers. The 8051 has fewer and smaller registers, leading to much higher usage of the individual registers in programs written for it, so it would be more sensitive to faults than a RISC-V processor.

1.6 Structure of the report

The remaining part of the report is divided into the following chapters:

1. Chapter 2 – Background – describes the basic concepts behind the topic of this project

2. Chapter 3 – Design – delves into the details and rationale of design and implementation of this project
3. Chapter 4 – Evaluation – is about the methods used to analyse performance, cost, and verify the correctness of the implementation based on experimental results
4. Chapter 5 – Conclusions – contains a summary of the results of this project

Chapter 2

Background information

2.1 Hardware design in Verilog

Most modern digital hardware is designed digitally using Hardware Description Languages, such as Verilog [06] or VHDL. These languages were designed to model and simulate hardware behaviour, and then synthesis tools were created that can turn those models into real hardware, be it silicon chips or configuration files for reconfigurable hardware.

This work focuses on Verilog, because it is one of the two main HDLs in use today, and because it is taught at this University. Verilog resembles the C programming language in many aspects, however because it models hardware, many aspects can seem counter-intuitive for someone with a software engineering background. The most important points to keep in mind about Verilog are as follows:

- The code is organized into modules, each module has some of its own logic and can instantiate other modules
- "Variables" are split into two main kinds, `wire` and `reg`
- `wire`s are generally used to model combinatorial logic, while `reg`s can be used for both combinatorial and sequential logic, dependent on where they are used
- Most logic is put into `always` and `assign` blocks
- `always` blocks can be combinatorial or tied to one or more edge triggers for modelling sequential logic
- A `reg` can be a memory component (like a flip-flop or a RAM) if it's assigned to in a sequential `always` block

Once a design is written in Verilog, it can be simulated to test the behaviour of the circuit, or it can by synthesized into hardware. The process of synthesis can take many steps and be very slow, especially if the target is to manufacture an integrated circuit (ASIC). However, an intermediate stage between real hardware and software simulation can be to tar-

get an FPGA – field programmable gate array. It's a reconfigurable circuit consisting of configurable logic blocks (CLBs), blocks of memory, sometimes specialized signal processing circuitry, all of those connected by a flexible interconnect. This architecture allows to test hardware designs with much higher performance than software simulation, exploit massive parallelism, without incurring the costs of silicon manufacturing.

2.2 Radiation and electronics

An important concern for many applications where hardware is used is the influence radiation can have on electronics. In space and nuclear deployments, very high levels of natural and artificial radiation can cause devices to malfunction frequently if no counter-measures are taken. Even in terrestrial applications, despite the shielding the Earth's magnetic field provides, radiation can influence the results of computation from digital circuitry. For example, there is a case of a (most likely) cosmic ray influencing the results calculated by an election results processing system [Joh17].

The radiation-induced errors can be separated into soft and hard errors (and long-term damage from the total radiation dose, but this is not related to this project). Soft errors are often Single Event Upsets (SEUs), where an ion causes a memory cell to flip its state from a 0 to a 1, or vice versa. This usually remains the case until the next write to that memory cell changes the value stored again, but the upset may have already propagated and caused wrong results in other areas of the system. Single Event Transients are another kind of soft error, when a charge triggers a signal line in a circuit for a brief moment. Hard errors include Single Event Latch-ups, which can result in a CMOS component permanently stuck in one state until a full power cycle is done. There are also other, more severe errors such as SE Burnouts and Ruptures that cause permanent and irreversible damage to the circuit.

While there are certain manufacturing techniques to minimize the effects of such faults, often shielding circuits can be more expensive or just infeasible compared to hardening the logic against potential failures, by e.g. introducing error correction codes. Especially for space missions, each gram can add a significant cost to the rocket launch, and shielding from radiation usually involves thick layers of heavy metals. That's why often a combination of certain shielding techniques is used together with digital hardening, to provide high reliability at a lower cost.

2.3 Hardware verification methods

In hardware engineering, as in any other engineering discipline, high confidence in correctness of work is desired. To achieve this, various verification methods are used. Some of the major techniques for HDL-based designs are as follows:

- Simulation – by writing "testbenches", the designs are virtually connected to drivers that give the circuit a pre-determined input, and monitor the output to verify that it matches expectations.
- FPGA testing – as above, but the testbenches are implemented for FPGA hardware to allow the circuit to run a lot faster, at the cost of setup time which is usually much longer due to synthesis tools.
- Mathematical proofs – for simple designs, a lot of properties can be proven "on paper" using standard mathematical techniques.
- Formal verification – this is a technique that uses automated reasoning methods to prove specified properties by exploring all possible states of the designed system.
- Code review – just as in software engineering, another engineer(s) inspect the design for potential faults manually.
- Self-test circuitry – for larger designs an extra mode is implemented, that allows the final design to test its various components to catch potential errors that occur at the manufacturing stage.

Because of the high cost of integrated circuit production a combination of all the above techniques is often used to achieve maximum confidence in the reliability of the design before it is set in stone (in this case silicon). This project assists with simulation and FPGA testing of circuits against potential single event effects, that may occur e.g. due to radiation, by providing a tool to allow the verification testbenches to test circuits with random, controlled faults injected.

2.3.1 Verilator

An interesting tool for Verilog verification is the open-source Verilator[Sny20]. Instead of directly simulating Verilog or compiling it to some sort of bytecode or executable foramt as a lot of simulators do, it generates C++ source code that matches the behaviour of Verilog. This can be used to construct testbenches in C++ rather than Verilog, for much easier integration with external libraries. The authors of Verilator also claim it is significantly faster than a lot of commercial simulators.

This project mostly uses Verilator's front-end, which is accessible through command-line options. Design elaboration – the phase of calculating the entire module tree and calculating parameters for their instantiations – can require partial Verilog evaluation. This is a complex task, requiring essentially a Verilog compiler implementation, so this task is deferred to Verilator in this project. Instead, Verinject uses the "AST" XML output of Verilator, which is a lot more than just the abstract syntax tree. It provides a breakdown of the hierarchy of modules, which was necessary for generating fault injection points correctly.

Chapter 3

Design

This project is split into multiple components, that all work together to realize the goal of verifying hardware designs in presence of single event upsets. The three main modules are as follows:

- Verinject — the Verilog source transformation tool written in the Rust programming language, it adds fault injection points to the designs
- Supporting Verilog modules — these modules drive the generated inputs in the hardware design to control where and when the faults are injected, and to report the injections
- Utility scripts — Python scripts that read metadata generated by Verinject and allow the user to precisely tweak how faults are injected and to understand the reports from the Verilog modules

This split came naturally as a consequence of the process of verifying designs and the desire for modularity of the system. The separation of the modules from the source transformation tool allow different tests to be performed by simply swapping just those modules, leaving the rest of the design untouched. Similarly, the utility scripts allow for more flexibility in both setup and reporting phases of the experiments without having to change or unnecessarily complicate the Verilog modules. Each of these components is described in detail in the following sections.

For additional context, a rough workflow to verify a Verilog design for error rates in presence of injected faults is as follows:

- Run Verinject on the source to make sure all features used in the source are supported
- Certain forms of syntax from previous standards and code-generation features might not work and need to be excluded or rewritten, or Verinject needs to be extended
- Re-run verinject on the adjusted source, it will generate a copy of the entire hierarchy with fault injection

- (Example test) Write a testbench instantiating the uninjected and the injected design, driving inputs and tracing outputs
- Adjust testbench to interface with the appropriate – simulation or AXI slave – driver module
- Set up simulation or synthesis to include the design with and without injections, and the desired driver and injection modules
- (Optional) Adjust FIFO depths on memory injection modules depending on expected experiments
- Generate traces with the `vj-gentrace` script
- Run experiments, logging output from the console or through the MMIO interface of the AXI slave
- Pipe the output through `vj-filter` to annotate it with locations where faults were injected
- Analyse the output

3.1 Verinject — the Verilog code transformation tool

3.1.1 Overview

This tool reads the input Verilog modules, and modifies their source code in the following way:

- Add an input for the state variable controlling where faults are injected
- Make sure that input is propagated to all submodule instances
- Count all the memory (including register) bits in the modules
- Assign individually addressable bit numbers
- Instantiate injection support modules covering all bits
- Replace reads from injected variables with the values coming from the support modules
- Generate signals on writes to injected variables, to keep track when the modified values are overwritten

A simplified flowchart of the tool's operation is presented in figure 3.1.

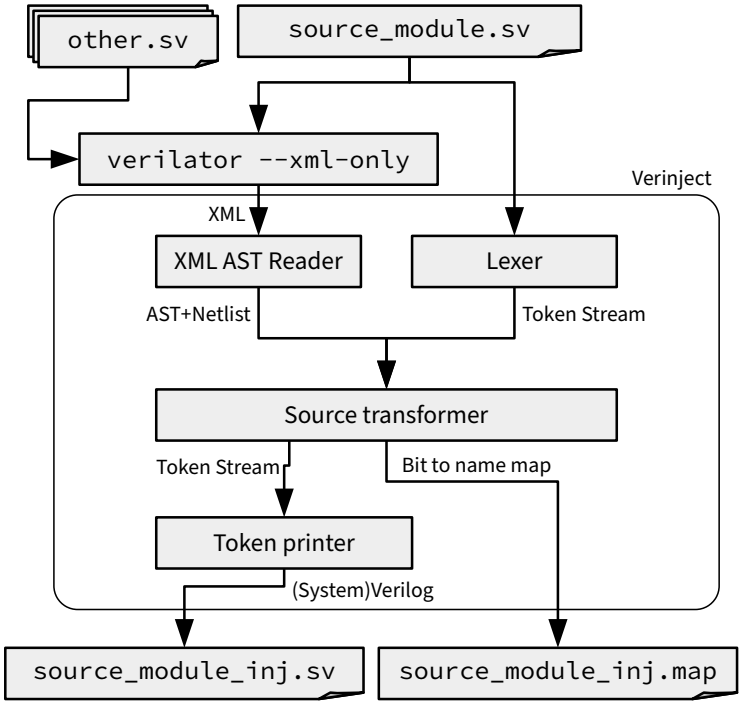

Figure 3.1: Source code transformation flowchart

Instead of creating a Verilog parser completely from scratch, this tool partially reuses the functionality of an open source Verilog to C++ compiler, Verilator[Sny20]. Another option was considered – the open-source Verilog synthesis framework, yosys. Unfortunately, as it is mostly a synthesis tool, a lot of information is discarded during the parsing process, and the parser was tied strongly to the rest of the codebase. This made it difficult to use it for source code transformation purposes required for this project. Verilator provides the tool with the final tree of modules based on the parsed sources, and detailed information about the types of variables and where they are used, in the form of an XML file. However, off-the-shelf parsers were not usable for the goal of source code transformation, as most of them lost information about the original files, such as whitespace, comments and some syntactic details that would be irrelevant to a compiler or synthesis tool, but are desirable in the output of Verinject.

The main reason behind the decision to keep the formatting of the files mostly intact is to make the output of the tool easy to understand by the authors of the unmodified source code, so that troubleshooting is easier. This was achieved by writing a custom lexer, splitting the source into tokens such as semicolon, identifier, parenthesis, etc., including whitespace and comments. Then the token stream was processed by a source transformer, which in many ways is similar to a typical parser, but it doesn't generate an entire syntax tree.

Instead, it uses a lot of information from Verilator's XML output and parses only the parts of the grammar that are needed to e.g. differentiate reads from writes. With that approach, it works through the token stream, either outputting the tokens unchanged, replacing some of them, or adds entire blocks of code e.g. for module instantiations. This led to a much simpler parser design, but it has the drawback of being less flexible in terms of extending it with functionality different than rewriting variable accesses.

3.1.2 Programming language choice – Rust

Verinject is written in the Rust programming language [20] for multiple reasons. Firstly, it is a systems-level programming language with the possibility to define C ABI interfaces, which could be useful if the tool needed to directly interact with Verilog simulators – C interaction is a part of the Verilog standard [06] in the PLI section. In contrast to using C directly, it provides strong memory safety guarantees which made it easier to avoid bugs in the program.

Rust also borrows concepts from functional programming, such as pattern matching and algebraic data types which make it much easier to express parsers and structure transformations than in C or C++. Subjectively, it is easy to install and can generate statically linked executables, so it should be easy to deploy both source and binaries of Verinject on hardware designers' machines. The language's strong type system, use of modules, relative similarity to C++ and SystemVerilog, and integrated documentation generator should also help maintainability and ease of access to the codebase for users potentially wanting to change the behaviour of the code.

3.1.3 Bit number assignment

One of the problems encountered in this project was to be able to uniquely address individual bits in the entire module tree of a Verilog design, so that faults could be injected precisely and repeatably. Related works used either Verilog simulators' APIs to access the simulation netlist containing a representation of all the components of the design, or worked on a synthesizer netlist level, which tied the implementation to a specific FPGA synthesis tool intermediate format. Those approaches make it easier to support a wide variety of designs, but tie the injection tool to a specific simulator or toolchain, making it difficult to e.g. switch from simulated injection to on-FPGA injection for increased speed of verification. They also make it easier to support Single Event Transients, which are faults injected in the middle of combinatorial logic rather than at memory components. This project does not attempt to simulate SETs – however in testing compared to real-world ion beam tests, adding SET simulation increased the accuracy from 84% to 87% [Eva+17]. This is a relatively small increase,

so it was determined to be less important than other goals of this project in its limited timeframe.

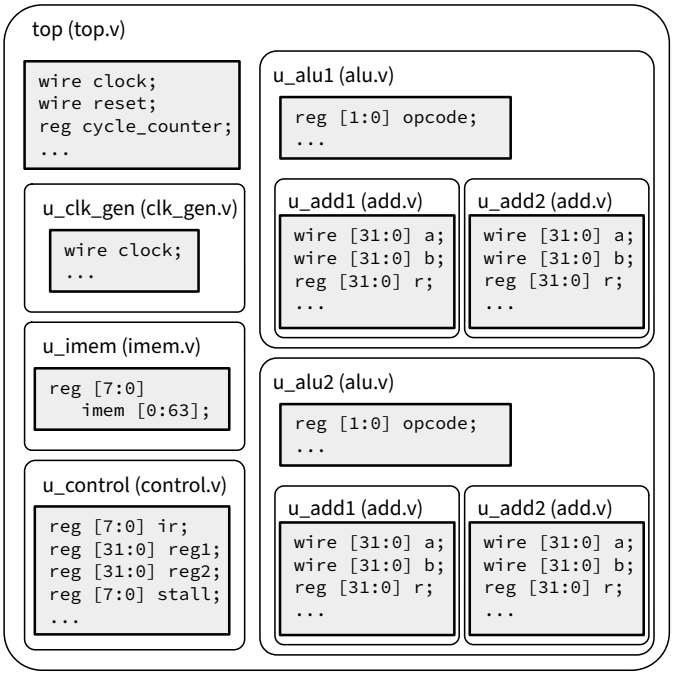

Figure 3.2: Illustration of a Verilog module hierarchy – connections between modules are not shown for clarity

For clarity of the following description, a sample Verilog module hierarchy is presented in figure 3.2. The module hierarchy forms a weakly connected and directed acyclic graph if we associate each module with a vertex, and each instantiation with an edge from the parent to the instance. The graph must be acyclic, because otherwise it would represent an infinitely large hardware design that's impossible to synthesize or simulate in finite time. It must also be weakly connected, because the top module must exist, and all other modules are discovered by traversing the instances present in module definitions, so no vertex is formed that is not reachable directly or indirectly from the top vertex.

In the current implementation of Verinject, different module instances are assumed to have the same internal structure (variable layout and child instances). This means not all parametrized modules are supported, this could be solved by duplicating the module definition for different parameter sets, but it was not done to simplify the problem for the scope of this project. This simplification leads to an depth-first search algorithm for assigning unique identifiers to every bit in a memory cell (clocked `reg` variables and Verilog arrays):

1. **For each remaining module**

2. Count how many bits are contained in its own variables – “own bits”
3. Recursively sum how many own bits and child bits there are in this module’s children
4. Store “own bits” + “child bits” as the total number of bits for this module

DFS can be safely applied because the module graph is acyclic, and it will cover all the modules used in the design thanks to the reachability from top property discussed above. One important thing to note in this algorithm is that modules can be instantiated multiple times in a single parent module. This is already handled by DFS as each instance is a separate edge, so all the bits get counted. Example total counts for the hierarchy presented previously in figure 3.2 are visible in figure 3.3.

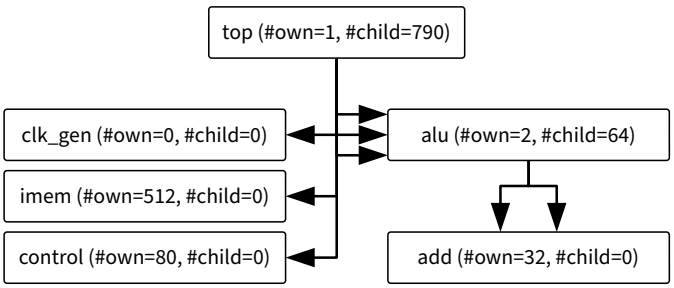

Figure 3.3: Numbers of memory bits calculated for the module hierarchy

Once those numbers are established, assigning bit identifier to modules is a fairly simple procedure:

1. **For each module M in the design:**
2. Add module parameter $start$
3. $count \leftarrow 0$
4. **For each** clocked variable and memory with b total bits
   (a) Add injector instance with bit identifier starting from $count + start$
   (b) Remember identifier range $[count + start, count + start + b - 1]$ for this variable
   (c) $count \leftarrow count + b$
5. **For each** direct child of M with c total own and child bits
   (a) Set the $start'$ parameter of the instance to: $start' = count + start$
   (b) Remember identifier range $[count + start, count + start + c - 1]$ for this instance

(c) $count \leftarrow count + c$

In summary, by exploiting the structure of the module instance graph, each module is modified such that its variables and children know the offset of their bit identifier from the parent's $start$ parameter. The parameter is a Verilog parameter, which makes this assignment translate in a direct manner to a Verilog source level transformation, and the actual sums of $start$ parameters and offsets are calculated at design synthesis or compilation time. Verinject also outputs a separate file with the identifier ranges for the whole module structure printed out recursively, so it's easy to match bit identifiers with their location in the design.

An example of what changes are done to the module's source code in this process is presented below:

```
// Before injection
module simple_multiple(input clk, input dat, output reg val);
reg [1:0] ff1; reg [1:0] ff2;
reg [1:0] mem [0:1]; reg ff3;

always @(posedge clk)
begin
  // logic assigning values to all above variables
end
endmodule

// After injection
module simple_multiple__injected #(parameter VERINJECT_DSTART = 0) (
  input clk, input dat, output val
, input [31:0] verinject__injector_state
);
// modified original variables and logic
// injection instances:
verinject_ff_injector #(.LEFT(1), .RIGHT(0), .P_START(VERINJECT_DSTART + 0))
 u_verinject__inj__ff2
( .clock(clk), .do_write(verinject_do_write__ff2),
  .verinject__injector_state(verinject__injector_state), .unmodified(ff2),
  .modified(verinject_modified__ff2)
);
verinject_ff_injector #(.LEFT(0), .RIGHT(0), .P_START(VERINJECT_DSTART + 2))
 u_verinject__inj__ff3
(); // similar ports
verinject_ff_injector #(.LEFT(1), .RIGHT(0), .P_START(VERINJECT_DSTART + 3))
 u_verinject__inj__ff1
(); // similar ports
verinject_mem1_injector #(.LEFT(1), .RIGHT(0),
 .ADDR_LEFT(0), .ADDR_RIGHT(0),
 .MEM_LEFT(1), .MEM_RIGHT(0),
 .P_START(VERINJECT_DSTART + 5))
 u_verinject_mem1_rd0__inj__mem
( .verinject__injector_state(verinject__injector_state), .clock(clk),
  .unmodified(verinject_read0_unmodified__mem), .read_address(verinject_read0_address__mem),
  .modified(verinject_read0_modified__mem), .do_write(verinject_do_write__mem),
```

```
39    .write_address(verinject_write_address__mem)
40  );
41  verinject_ff_injector #(.LEFT(0), .RIGHT(0), .P_START(VERINJECT_DSTART + 9))
42   u_verinject__inj__val
43  (); // ports similar to ff1,2,3
44
45  endmodule
```

In the code above, the $start$ module parameter is named `VERINJECT_DSTART` and the bit identifiers are the `P_START` parameters to the instantiated verinject modules. The order in which variables are processed is deterministic, but as it comes from a hashing function in the internal data structures it can be hard to predict.

3.1.4 Source transformation process

The process of transforming Verilog source code to add fault injection is split into several, partially intertwined, stages:

1. Read the module structure, variable list and hierarchy from Verilator's XML output – located at `src/xmlast.rs`
2. Generate bit number assignments as described in the previous section – in `src/xmlast.rs`
3. Split the input sources into arrays of tokens (including tokens for whitespace) – in `src/lexer.rs` and `src/transforms/mod.rs`
4. Detect important syntactic structures in the token array (with a recursive-descent parser) and add fault injection – in `src/transforms/inject_ff_errors.rs`
5. Generate a map file of all the variables in the hierarchy with their bit identifiers – in `src/transforms/generate_bit_map.rs`

The following transformations are applied to Verilog files:

- Add a `verinject__injector_state` port to the module, used for communication sent to the injection modules
- Remove the `reg` net type from clocked output ports and variables – as they'll be driven from the injection modules
- For each clocked variable's declaration, add an instance of `verinject_ff_injector` with appropriate parameters
- For each memory variable declaration, add an instance of `verinject_mem1_injector` for each read of the variable

- Modify each instance of child modules to include the `VERINJECT_DSTART` parameter set to the correct value
- Replace all reads (occurrences on the right side of assignments) of clocked variables with the outputs of the injection modules
- For memory reads, also add code to capture the read address for the memory injection module
- For all writes to injected variables, add code to capture the fact of writing to allow erasure of fault in SEU simulations

The transformations are applied only to variables that are assigned to in always blocks triggered on an edge of a signal, that way only sequential logic (memory components) are affected. Affecting combinatorial logic would require partial synthesis of expressions and injecting faults at gate level, which is outside the scope of this project.

3.1.5 Output formats

Verinject outputs two kinds of files: Verilog and its own map format. Input Verilog modules are expected to be in separate `.v` files, and it generates separate `__injected.v` files for each module, in a given output directory. The names are changed so that the two module hierarchies can coexist, for example for a simulation that compares the behaviour of the system with injection vs without. Alongside the modules, it outputs a `top_module_name.map` file that looks like this:

```
# [this header is not a part of the actual file]
# "Range start" "Range end" "Path.VarName" "Total bit count" "Left word range" ..
# .. "Right word range" "Left memory size" "Right memory size" "Kind: var/mem"
0 1 simple_multiple.ff2 2 1 0 0 0 var
2 2 simple_multiple.ff3 1 0 0 0 0 var
3 4 simple_multiple.ff1 2 1 0 0 0 var
5 8 simple_multiple.mem 4 1 0 1 0 mem
9 9 simple_multiple.val 1 0 0 0 0 var
```

The bit map file is designed for trivial parsing by other tools, it's a simple list of bit identifier assignments in consecutive lines. It has columns for the start and end identifiers of bits assigned to a given variable, and also the number of bits per memory word and the Verilog bit range used in a bit vector declaration to facilitate pretty-printing.

This file is used by the utility scripts to e.g. convert simulation outputs to a human-readable format including locations of injected faults in terms of Verilog modules. The same file can also be used to generate input injection stimulus by another tool – by black- and white-listing certain signals to generate a more controlled fault injection trace. Moving of this functionality into separate tools and restricting the format used by the Verilog modules to just single

numbers significantly simplifies the implementation of those modules, and reduces the size and complexity of hardware driving the fault injection.

3.2 Supporting Verilog modules

The fault injection implementation relies on a few hand-written Verilog modules, namely:

- `verinject_ff_injector.v` – handles the injection into non-array `reg` clocked variables
- `verinject_mem1_injector.v` – handles the injection into array `reg` memories with a single write port
- A signal generator module – drives the bit identifier signal that triggers fault injections on chosen clock cycles
- A signal monitor module – communicates the triggered fault injections back to the simulation console or device connected to the FPGA

Each of them is independent of the rest, so they can be swapped out in order to perform different tests. For example, there are testing implementations of the injector modules (in `verilog/test/`) that do not persist the bit flips, that can be used in conjunction with a test generator (in `verilog/gen/verilog_serial_tester.v`) that cycles through all possible bit identifiers. This is the way the bit assignment was verified to be propagated correctly throughout the Verilog modules, and that it matched the generated map files. Another feature it allows is an easy migration path from simulation testing to FPGA testing and back, as only the signal generation and monitoring modules need to be changed to provide a matching interface, and the rest of the design with fault injection can be left mostly unchanged. Of course, further modifications to the testbench would probably also be necessary, but assisting with removing non-synthesizable user code is entirely out of the scope of this tool.

3.2.1 Injector modules

Three injector module implementations are provided and have been designed from scratch during this project:

1. `verilog/test/*` – Injects a fault into the given bit only in a single cycle, good for debugging the tool while looking at simulator waveforms
2. `verilog/ff/` and `verilog/memory_fifo/` – Have persistent state that simulates how actual SEUs would behave in circuits

3. `verilog/selu/*` – Similar to the above, but don't reset the faults on writes to model SE Latch-ups and SE Latch-downs

The flip-flop (register) injection modules create a register of the same width that's used as an XOR mask for the input. This way, it can handle multiple bit flips in a single register, and naturally handles a second bit flip occuring at the same location, "undoing" the effects of the previous flip.

The memory injection modules work similarly, but instead of creating a copy of the entire memory block, they only store a specified (default of 4) number of bit identifiers into which faults were injected in a queue structure. This way the impact on resources used by large memories is minimal. This is important, because in FPGA designs huge memories are sometimes used in designs, and avoiding a 2x cost allows the designs with fault injection to run on cheaper FPGAs.

3.2.2 Signal generator and monitor – for simulation

For simulation, two modules are provided for fault injection driving and monitoring:

1. `verilog/monitor/verinject_sim_monitor.v` – A very simple module, monitoring the injector state signal for non-all-1s values and displaying them to the simulator console with `$display`
2. `verilog/gen/verinject_file_tester.v` – Contains a trace memory (loaded from a specified file) and cycle counter. It accepts traces from `vj-gentrace` in the form of a list of (cycle, bit id) pairs and generates the right injector state signal at the specified times.

The implementations of these are quite flexible thanks to their simplicity, and have been useful for the tests ran while evaluating the tool. The simplicity also allows them to be modified or reimplemented for experiments that may need integration with specific IP or proprietary simulator APIs for richer information.

3.2.3 Signal generator and monitor – AXI slave for FPGA integration

In order to allow the fault-injected designs to run on FPGAs, a synthesizable interface needed to be implemented. Because of the hardware available in the School – Tul PYNQ-Z2 FPGA boards based on Xilinx Zynq-series, the FPGA signal generator has an AXI interface[Arm20]. AXI is used in both Xilinx and Intel (previously Altera) FPGA boards with on-board Arm processors, so this should cover quite a wide array of devices. Instead of designing an AXI slave interface from scratch, it's based off an example design from Dan Gisselquist's bus bridges repository[Gis20] released under the Apache 2.0 open source license. This design was easy

to extend with custom registers and memory blocks exposed to the host Arm processor, and it had already been verified in other designs and using formal techniques.

The Verinject module extends the AXI slave module to expose a trace memory, similar to the one used for simulation, but reprogrammable from the Arm side. It also has a simple logging interface for the attached testbench and a cycle counter with a MMIO-controlled reset switch for the device under testing. This set of features was enough to run the experiments in this project, and if more ports were needed for another application it would be minimal effort to extend it. An example usage block diagram for the array addition FPGA experiment (as described in the Evaluation section) is shown in figure 3.4.

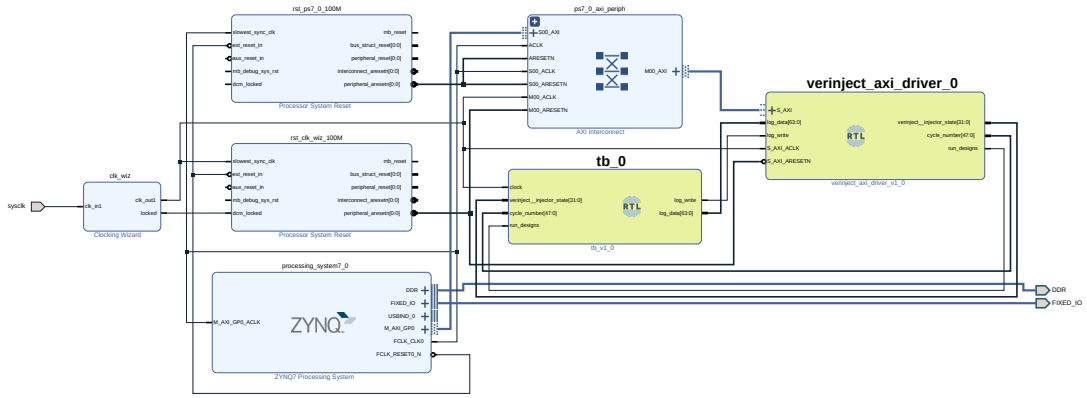

Figure 3.4: Block diagram for a Xilinx Zynq testbench for the array addition experiment

The interface on the Arm side is exposed via a simple C header file (located in the same directory as the Verilog module). The header defines enums for the bitfield values and a struct representing the MMIO interface, allowing access to the trace and log memories as arrays, and other control registers as simple members.

3.3 Utility scripts

Alongside the source transformation tool and Verilog modules, some utility Python scripts are included for convenience. Python was chosen because of its widespread use for scripting, so it should be available on any machine used by this tool's users, and its standard library is large enough that the scripts would have no external dependencies. The scripts, with their purpose are as follows:

- `vj-filter` – filters the simulation output log to include fault locations based on a map file; named similarly to a gcc utility, `c++filt`, which demangles C++ symbols in console output
- `vj-gentrace` – generates a stimulus file with a given random seed, and optionally a list of allowed or banned variables to control where faults are injected

They are simple in design and implementation, and provide enough flexibility to be used directly, through bash scripts, or modified to suit an alternative purpose.

Chapter 4

Evaluation

In order to verify Verinject yields correct results, can be trusted to be used in validating hardware, and to determine fault injection's overhead, multiple approaches of testing were combined:

- Unit testing — small examples of Verilog code are present in the test folder of the source code, at least one for every major supported feature.
- Manual inspection on a simple sample — was used to confirm correct bit id assignment and triggering by inspecting simulation waveforms.
- Run on a mathematically-analysable example — to show that the behaviour of a well-known circuit under fault injection matches predictions.
- Run on two real-world processor designs — to show appliacbility to real-world scenarios.

These approaches are discussed in the following sections.

4.1 Unit testing

The Verilog language is very expressive, and with that follows a complex syntax. Therefore, to support enough features for Verinject to be usable, it was important to write tests verifying that they can be used individually in the first place, before testing more complex cases. For this purpose, a set of small, simple examples was written, with a script that runs verinject on all of them and checks if the output passes the Verilator linter with no errors or warnings.

This approach helped in keeping track of progress in early development, and in finding regressions which inevitably happened as the project's code evolved. Some of the cases also aided in debugging some of the errors encountered with more complex designs – by reducing the complex errors to the minimal example triggering the bug, more tests were created.

The linter pass/fail approach did not catch any logic errors in the modified code, so it was im-

portant to manually inspect the output on major code changes. The simplicity of examples and the fact that Verinject tries to preserve a lot of formatting – like whitespace and comments – in the generated output made the manual inspection a lot easier.

The goal for these tests was not to exhaustively test the correctness of the project, or to catch a majority of bugs, but to aid in debugging and to prevent simple bugs. The unit tests were successful in this aspect, and a very valuable tool in development of this project, but would not suffice as the only way of evaluating success.

4.2 Waveform inspection

One of the key features that needed verifying was the bit identifier assignment. It's core to the functionality of Verinject – because it controls where faults are injected. A failure in correct mapping of bits to memory components would make analysis of randomly injected faults much more difficult and the tool would not be reliable for performing reproducible experiments. The guarantee that needed testing specifically is: for every request to inject a fault at a bit with the given identifier, the identifier propagates correctly through the fault injection mechanisms and leads to an injection in the correct bit.

In order to test this hypothesis, a simple, tractable Verilog module was written (present in `samples/01_bitstest`). It contains some memory components of varying sizes, and performs writes to them with easily predictable values such as all zeros, or a Boolean negation of another value. Fault injection is ran on this module with a slightly modified driver: the persistence of faults is disabled, so they only happen for one clock cycle. On top of that, the bit identifier is cycled through all its possible values, to test every bit in the module. These modifications allow for the entire module to be tested quickly, without having to reset the entire circuit after a full test cycle for another bit identifier.

This prepared circuit was run through Icarus Verilog, an open-source Verilog simulator (any other compatible simulator could be used) to produce a set of waveforms of all the signals present in the system. Then, a manual inspection was required to confirm that the identifiers generated corresponded to faults in the right places based on the mapping file and the assignment algorithm "ran" on paper. This verified that the identifiers matched the right bits in the right memory components.

4.3 Array addition experiment

A first bigger test of the project was the array addition experiment. The goal was to verify that the behaviour of a known circuit under fault injection would match theoretical expect-

ations. In order to make this problem easily tractable, a simple circuit was chosen: it added numbers stored in two memory blocks together index by index, and output the sequence of sums to a port. The core of the logic is presented in the following Verilog listing (skipping the declarations and initialization for conciseness and clarity):

```
always @*
begin
  index_nxt = index_r;
  if (run)
  begin
   index_nxt = index_r + 8'b1;  // loops around on overflow
  end
  word_a = memory_a[index_r];
  word_b = memory_b[index_r];
  sum = word_a + word_b;
end

always @(posedge clk, negedge rst_n)
begin
  if (!rst_n)
  begin
    index_r <= 0;
  end else if (run)
  begin
    index_r <= index_nxt;
  end
end
```

4.3.1 Measurables

The measurables we can reason about and that yield interesting results are as follows:

1. Dynamic cross-section: Probability of a fault injected into one random bit affecting the output of the circuit during its entire run at all
2. “Cascade effect”: Distribution of the number of bits in the output changed across trials – in every trial only one bit in the circuit is flipped
3. Time to perform one trial: measures what the impact of fault injection is on the runtime of simulation and FPGA trials
4. Cost in logic elements on an FPGA: measures the resource cost of introducing fault injection compared to the base circuit

Even though this is a very simple circuit, all the above effects will be present in bigger, more complex designs such as processors. However, in the case of this vector addition experiment they can be predicted with relative ease, so the injection methodology can be verified before applying it to designs that can’t be analysed mathematically.

4.3.2 Dynamic cross-section

4.3.2.1 Theory

Not every injected fault will affect the computation performed by a digital circuit. For example, in many computer programs there will be sections of code that don't use the value of a particular register (it is "dead"), after which the register will be loaded with a fresh value. If a non-permanent fault injection occurs in that register while it is "dead", the computation won't be affected at all. Another example likely to happen in a real system is if the most significant bit of a register gets latched to a zero, and the register is only used for small, positive integers. In that case the impact of the fault won't be visible to the running programs. Velazco, et al. [VRE00] call these errors "tolerated errors".

In nuclear physics, the probability that a particle in an accelerator interacts with the target is proportional to a characteristic parameter of the target, called the cross-section. Hence the name dynamic cross-section of a program or circuit – it is the fraction of bits located in time and space that under a fault injected will affect the results. This dynamic cross-section can be used together with a static cross-section (of the physical components making up logic gates and memory cells) in the nuclear reaction rate equation to estimate the error rates in a physical system in a pre-determined environment. The simplest form of the equation is that the number of reactions per time per volume is equal to the product of the particle beam flux and the cross-section of the system.

For our system, estimating the dynamic cross-section analytically is very simple. If a bit flip is injected into any bit of either of the source matrices, the result will change if, and only if, the sum using the particular bit has not been calculated yet. Another possibility is an injection into the index register, which will start adding completely different values, which will also affect the output seen from the chip. For the first case, we expect the probability of an error leading to a change in the output to be $\frac{1}{2}$ because the only factor at play is whether the modification to a bit occurs before or after (in time) the calculation of the sum for that particular index, and the time of injection and the index to which the fault is injected are two independent, uniformly distributed random variables. The easiest way to visualise this is with a table showing whether an injection at a particular word, and particular time leads to change in result (✓) or not (×):

Word	$t \in [0, 31]$	$t \in [32, 63]$	$t \in [64, 95]$	$t \in [96, 127]$
W0	✓	×	×	×
W1	✓	✓	×	×
W2	✓	✓	✓	×
W3	✓	✓	✓	✓

Because both the word index and time are uniformly distributed, for a large number of words the probability is approximately $\frac{1}{2}$ as read off the above table. For many words, the potential faults injected into the index register become unlikely – for n b-bit words there are nb bits in the word memory, and only $\log_2 n$ bits for the index register – an exponentially smaller number. Therefore it's expected for the circuit under fault injection to yield a different (wrong) result in slightly more than half of the cases with faults injected.

4.3.2.2 Experimental results

In order to verify this theory, the circuit was run through a testbench that compared the results from a reference version with the results from the version with fault injection added by Verinject. The test was repeated 100000 times with different random seeds for the fault injector, and all discrepancies were stored into separate files for easy classification. The test circuit had two arrays of 256 32-bit words each added together, with an 8-bit index register.

In the results, in 50448 of 100000 cases there was a fault in the output and in the remaining 49552 cases there was no fault reported. This agrees with the theoretical predictions from the previous sections, giving a dynamic cross-section of 50.5% – slightly higher than half, as expected. The probability of the fault occurring in the index register (given that a fault is occurring somewhere) is $\frac{8}{8+2*256*32} = \frac{8}{16392}$ which is 1 in 2049, so the expected number of cases for this would be 49, and in the 100000 cases it occurred 45 times – again matching the theory.

This test was also used to perform a validation of the FPGA implementation of fault injection. The same design was run on the FPGA with the AXI adapter connected to the on-board Arm processor. The disparities between the two circuits were logged on the FPGA, and read back from C using the memory-mapped interface of Verinject's AXI adapter, then sent back to the host PC over an UART serial port emulated over USB by the board. The same fault injector configurations were used to check if the faults are the same between simulation and hardware.

The comparison of the faults recorded showed, that the FPGA version showed the same faults as the simulated one, and the faults were in the same place, time (cycle), and the differences in the output from the tested circuit were exactly the same. This confirms that Verinject can be used portably and simulations can be turned into FPGA tests relatively easily, allowing for more tests to be performed in a shorter time (this is described in the Performance subsection). And with a failing FPGA test, the same settings can be used in the injector to reproduce the failure in a simulator, allowing for much easier troubleshooting thanks to the simulator's ability to visualise all signals in the design.

4.3.3 Cascade effect

4.3.3.1 Theory

Another interesting metric to look at is how a single fault in a single bit can "propagate" and cause multiple faults in the output. It might seem obvious, but it is important to note that logic and memory components are interconnected, and if a change in one of them matters for the output, it probably affects multiple other components in the system. This can be seen even in this very simple addition circuit, for example:

- Let's take the binary sum $0011\,1001 + 0001\,0010 = 0100\,1011$
- If we flip the rightmost bit in the sum: $0011\,1001 + 0001\,001\mathbf{1} = 0100\,1\mathbf{100}$, one bit changed in the input led to three bits in the output changing
- However, if we flip the leftmost bit: $\mathbf{1}011\,1001 + 0001\,0010 = \mathbf{1}100\,1011$, a one-bit input change leads to a one-bit output change

This behaviour is caused by the carry chain in addition – flipping one bit can cause many others to flip if it changes the carry propagation behaviour.

To predict how the bits may change, we have to look in detail at how the carry process in binary addition works. Because in this project faults are not injected into combinatorial logic, unless an adder is pipelined, we can assume that any adder circuit will behave the same when a fault is injected into its input, as they all would produce the same result. Therefore, we can use a simple ripple-carry adder as our mathematical model for addition behaviour. Let's take a look at the truth table for a full adder component (adding two bits and a carry-in):

Row	a	b	c_{in}	c_{out}	s
0	0	0	0	0	0
1	0	0	1	0	1
2	0	1	0	0	1
3	0	1	1	1	0
4	1	0	0	0	1
5	1	0	1	1	0
6	1	1	0	1	0
7	1	1	1	1	1

First important observation is that any change in the $c$arry-in signal leads to a change in the $s$um signal – which means that any carry-in change in a sum would lead to at least one extra change in the resulting sum on top of the less significant bit flipped in the first place. To predict how often this will happen, we need to estimate how often an extra carry will be generated.

The fault can affect only one of the three inputs to any full adder, as we assume there is only a single fault injected in this analysis. If it's in a or b, this is the first failing bit, otherwise if c_{in} has a fault that means a fault has propagated from the previous bit already. Because the adder computes a sum, which is associative and commutative, we can assume without a loss in generality that the fault occurs in a – if it occurs in any other input, it's equivalent to swapping that input with a, and the result would be the same.

With these facts and assumptions, we can now obtain the probability that a bit flip in one of the inputs to the adder will propagate to the next adder in the carry chain. Let's call this probability P_p. A fault will propagate if a flip in a leads to a flip in c_{out}. This occurs for rows 1, 2, 5 and 6 – 4 out of 8 possibilities. Therefore P_p is $\frac{1}{2}$.

We can now combine this knowledge with the base probability of a fault affecting the result at all from the previous section. Let $P(n)$ be the probability that n bits change in the output upon a single random fault injected into the input to the adder. $P(1) + P(2) + \cdots = \frac{1}{2}$ because that's the probability that a change will occur at all. This is mutually exclusive with a fault not occurring, so $P(0) = 1 - P(1) + P(2) + \cdots = \frac{1}{2}$.

To calculate $P(1)$ let's use the fact that it means there is a fault but it won't be propagated. Propagation is conditional on a fault or the previous propagation, which lets us multiply the probabilities in a simple fashion:

$$P(1) = P(fault) \times P(not\,propagated) = (1 - P(0)) \times (1 - P_p) = \frac{1}{4}$$

By repeating this reasoning with one propagation, we obtain:

$$P(2) = P(fault) \times P(propagated) \times P(not\,propagated) = (1 - P(0)) \times P_p \times (1 - P_p) = \frac{1}{8}$$

By induction, this leads us to a geometric series of probabilities $P(n) = \left(\frac{1}{2}\right)^{n+1}$. We can see this set of probabilities is complete for an infinite number of added bits, as the sum of this series is $S = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots = 2S - 1$, so $S = 1$.

Adding a boundary condition that faults can't propagate at the end of the word size would only slightly affect this distribution of probabilities, and finding an approximate geometric series in experimental results would be a good confirmation of the theory working without getting more precise estimations. The main property of geometric series is that the ratio between consequent elements is constant, so that's the property to look out for in the results.

4.3.3.2 Experimental results

When the injection was performed in the previous section, not only the presence of faults was logged, but also the XOR of the sum with fault injection and the reference one. Therefore

counting how many bits were changed in every example is as simple as looking at the number of ones in the binary representation of the XOR value. The summarised results for the same 100000 experiments are presented in figure 4.1 and table 4.1.

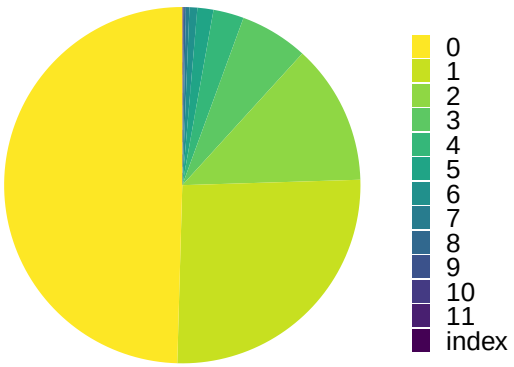

Figure 4.1: Pie chart of the measured distribution of the number of bits affected in the output when a single fault is injected

The ratio between consecutive frequencies remains fairly constant for n up to 9, above that the small probabilities and number of trials limited to 100000 lead to less accurate results. However, the experimental probabilities are reasonably close to the expected ones derived in the previous section. Therefore, the circuit under Verinject's fault injection performed as expected in the analysis before.

It's also visible that the probabilities for $n > 2$ are always slightly lower (the inverse probabilities in the table are higher) than their expected value. This is most likely due to the finite size of a word in this circuit – 32 bits. When a fault is injected into e.g. the 30th bit, it is impossible for it to carry into more than 3 bits. Generally, the shorter the word, the less likely higher n-s are in this distribution, which explains the observed behaviour.

This behaviour is interesting to study not only because it confirms that the fault injector developed in this project works as intended, but also because it shows on a simple example how faults can "cascade" through real systems. Reducing this effect, for example by isolating components in a fault-tolerant system, can yield a much safer system than one where fault from any subsystem can propagate to any other subsystem. Example ways of achieving this include triple modular redundancy (TMR) – where a given subsystem is duplicated three times, and some extra voting logic makes sure faulty computation doesn't affect any other module. Another possibility are error correcting codes, such as SECDED, used in con-

n	Cases	Ratio to previous cases	Experimental $P(n)^{-1}$	Expected $P(n)^{-1}$
0	49552	–	2.02	2
1	25935	0.52	3.86	4
2	12749	0.49	7.84	8
3	6181	0.48	16.18	16
4	2773	0.45	36.06	32
5	1455	0.52	68.73	64
6	713	0.49	140.25	128
7	322	0.45	310.56	256
8	155	0.48	645.16	512
9	71	0.46	1408.45	1024
10	47	0.66	2127.66	2048
11	2	0.04	50000.00	4096
in index	45	–	2222.22	2049

Table 4.1: Simulation data for the “cascade effect” experiment

junction with e.g. memory scrubbers that periodically remove faults from a module by using redundant data to reconstruct the state from before a fault occurred.

4.3.4 Performance

Another aspect this design allowed to evaluate is the performance of this fault injection implementation. The design was run in the open-source simulator Icarus Verilog, and on a Xilinx FPGA-based PYNQ-Z2 board. Times were measured to run the first 1000 tests from the set used in sections above, each case with a different but consistent random seed. In order to see the impact of fault injection, a second set of tests was run with the “injected” module in the testbench replaced by a second copy of the “uninjected” module. The simulator was running on 1 core of an Intel i7-6700HQ processor with 16GB of RAM available. The results are presented below:

Platform	Fault injection added	Time to perform 1000 tests (seconds)
Simulator	✓	115 (26% longer than without injection)
Simulator	×	91
FPGA	✓	3.04

The FPGA tests with fault injection are almost 38x faster than the corresponding simulator tests, which shows how much time can be saved with FPGA testing. For much larger designs, such as processors, the difference may be between being able to run the tests before ship-

ping a product, and not. Therefore, the flexibility provided with Verinject is useful – it allows to move to a higher-cost platform for higher performance without radically changing the fault injection procedure.

The FPGA testing time was constrained largely by the bus speed from the on-board Arm processor to the host computer, so to measure the impact of fault injection on the speed of the resulting circuit another measurement can be used: shortest clock period possible for this circuit. Synthesized with Xilinx Vivado with relatively aggressive optimizer settings, the testbench with fault injection was limited to a clock of 9.717ns, and without injection it only decreased to 9.558ns (1.6% change). This is not a very large impact, thanks to the very small overhead of the injection modules, however it might be higher for more complex designs.

The testbenches were also not excessively optimized – obvious bottlenecks were removed, but theoretically it would be possible to get more performance out of this circuit with a lot more work. However, this is more representative of a real situation, where there usually is a limited budget for optimizing the testing infrastructure specifically.

4.3.5 Logic element cost

Verinject requires roughly double the "small" memory components (non-arrays), and a constant amount of memory for each larger array. This means it should scale roughly linearly in terms of logic element cost for FPGAs and not require a lot more RAM components. This can be seen in the logic element costs for this design, as summarized in the following table:

Component	Used with injection	Used without injection
Look-Up Table	3325	1856
LUTRAM	713	713
Flip-Flops	2013	1608
Block RAMs	2.5	2

It's clear that there is a large cost to adding fault injection to the design, however it is less than double in this case. Therefore existing FPGA testing infrastructure can be either utilized without a change or would just need to be expanded 2x to cover the requirements for fault injection with Verinject.

4.4 Processor design test

The previous tests were relatively small-scale, useful for troubleshooting and proving parts of the project worked as intended. A logical next step was to test Verinject on a much larger design – such as a processor. Initially the Intel 8051-compatible core [TS01] was chosen,

because an 8051 processor was previously tested by the NETFI project [MV13], and with an ion beam in hardware [Rez01]. Unfortunately, the core was written in Verilog before the 2001 version, which is the primary target of Verinject, so a lot of code needed to be adapted to use slightly newer syntactic structures.

The 8051 Verilog code passed Verilator and Icarus Verilog lint checks after the changes, and the code generated by Verinject based on that also passed those checks – which was the first success. However, in simulation, the author was not able to get the original core (even before modifications) to correctly run 8051 programs. Simple changes, such as adding a NOP before the instruction, were changing the behaviour of the entire processor. After spending a few days on troubleshooting, instead of reverse engineering the entire design made in 2001, the author changed focus to use a more modern processor design instead. The RISC-V core used in the Computer Architecture and Design course was chosen.

Adapting its source for Verinject was a much simpler task, it only used a couple of unsupported Verilog syntax features: a couple of for loops, and preprocessor usage for module instantiation. Both of these were very easy changes, by just manually unrolling the loops and replacing a preprocessor define usage with the definition's value itself. A few more complex statements confused the Verinject parser, and required putting `begin`/`end` blocks around them, but no other major issues were encountered. When the modified processor design was confirmed to run simple programs correctly in simulation, fault injection code was added. The further modified core was also confirmed to run those programs, without any faults actually injected yet.

With a working processor with fault injection, a program for testing was needed. In order to stay close to the original goal of verifying the results against previous research work, a 6x6 integer matrix multiplication program was written in C, performing the same operation as the radiation-tested 8051 processor [Rez01] (table 2.11 in that thesis). The results for the multiplication were verified with a similar C program ran on the author's x86 laptop, to make sure the program was working correctly.

Fault injection enabled for the entire design would have led to results incomparable with the radiation testing, as the RISC-V core has many more registers, and most imporantly a lot more memory than the 8051. The 8051 had 128 bytes of total RAM, and about 24 8- and 16-bit registers. The RISC-V core has multiple kilobytes of RAM, split into instruction and data memory, and 31 32-bit general purpose registers, plus more special function registers.

In order to see the impact of these additional (mostly unused) memory cells, the experiment was repeated with three areas for fault injection: whole design without the branch prediction and instruction memory as the first area, second like the first, but also excluding the

mostly unused data memory, and third was just the general purpose registers. The results for 1000 trials for each of these areas are summarized in table 4.2. The instruction memory was not included, because the program was very small, and it could be treated as a read-only memory anyway, if it was for example Flash memory, it would be immune to radiation effects. It is the most likely area in a real processor to be hardened against radiation due to how critical it is.

Tested area	Dynamic cross-section
Without ICCM&BP	4.6%
Without ICCM, BP& DCCM	13.0%
Register file	23.1%
8051 ground radiation testing [Rez01]	46.71%
8051 simulation [Rez01]	50.05%

Table 4.2: Dynamic cross-section for the RISC-V processor performing a 6x6 matrix multiplication

It's clear that the results are different from the 8051, most likely due to the larger number of unused memory structures in the RISC-V processor for this simple program. The results are, however, reasonable – leading to the belief that Verinject works correctly. When injecting faults just to the register file, the measured dynamic cross-section (explained in section 4.3.2.1) is only half of the one measured for the 8051. Looking at the disassembly of the C program for the RISC-V processor, only roughly half of the available 32 registers were used, and because the 8051 has fewer registers to work with, the program in [Rez01] likely used all of those. This can explain the difference for this experiment. It also means that RISC processors, which generally have more registers, may be safer for operation in high-radiation environments than simple embedder processors, simply because it's less likely a critical component would get hit by an ion. On the other hand, they would have greater total surface area (static cross-section), meaning a higher probability of getting hit at all, so this might balance out the gains from a smaller dynamic cross-section.

Chapter 5

Conclusions

In conclusion, during this project a working Verilog source code transformation and fault injection was created. It can be used as a part of the process of designing hardware for use in environments that require fault tolerance. Verinject – the tool developed – helps in identifying areas for stronger fault tolerance and to debug issues with handling random faults. It's more flexible than a lot of other existing fault injection tools, because it is not tied to any particular vendor's toolkit, instead it operates directly on Verilog source files. The tool is released under an open-source license at `https://github.com/kubasz/verinject`, with certain copyrighted sample code removed from the public repository (such as the RISC-V CPU core).

The tool has a modular design, with the source modification, fault injection and fault reporting in separate modules with clearly defined interfaces. This allows for modifications to be easily made to adjust it to a particular use case. Simulation and FPGA interfaces for fault injection control have been developed, and confirmed to work in practice. The generation of test stimuli is achieved with a relatively simple script, with enough configuration options to satisfy a lot of needs, but thanks to the modularity it's also easy to program in new capabilities.

A major problem solved in this project is efficient (in the sense of little overhead) fault injection into HDL-based hardware designs without tying the injection to a particular simulator API or FPGA configuration format, like many previous works [SBB19] [MV13] [Eva+17] did. It is also more efficient than the FITO [SM08] project for designs using large memories, by not requiring decomposing them into individual flip-flops. This was done by designing algorithms working on the HDL specification level of a circuit, rather than final gate configuration, while taking into account the fact that certain HDL structures correspond to much more expensive hardware structures. This approach was more difficult, but led to a far more flexible solution in the end. The difficulty cost unfortunately also meant that not all features of Verilog were implemented in the timeframe of this project, but enough were done that it was possible to use Verinject on real-world designs by spending very little time modifying

them.

A new whitespace-preserving parser for Verilog was developed out of necessity, but a large amount of information is parsed by the widely-used open-source Verilog simulation tool, Verilator [Sny20]. Preserving the rough formatting, layout, identifiers and comments in a file helps significantly in debugging problems using this tool, as a major intended use case is to reproduce a faulty test case in simulation to find the root cause of a problem. Bits to inject faults into are addressed by identifiers generated from a predictable algorithm, allowing for consistent simulation and hardware behaviour of the code with fault injection. Assigning these identifier based only on the source of the module hierarchy was a challenge, and an algorithm for doing this based on the graph properties of the Verilog module structure was developed and presented.

Verinject was tested using a variety of different methods. It was run on a set of short syntax examples to ensure it supports a wide variety of constructs in the language. On top of that, the bit identifier assignment was tested by manual inspection of a simple example. Much bigger experiments were also performed with an array addition circuit, and two processor designs.

The array addition experiment confirmed that the behaviour of faults injected by Verinject matches expectations from mathematical theory. For example, the dynamic cross-section of the design – the probability that a bit error in a random location and time would lead to an error in the output – was very close to the theoretical prediction. Moreover, this test allowed to measure the impact of fault injection on the design: a small, 1.6% maximum synthesized frequency impact and a less than double increase in logic component usage on an FPGA.

Memory block usage of fault injection components is minimal, thanks to the use of small buffers storing injected faults rather than duplicating entire memory arrays. Testing in simulation using Icarus Verilog was found to be 38x slower than equivalent testing on an FPGA, by running the same set of 1000 tests on both. This also confirmed that the results and faults injected between the two targets were the same – which allows for the use case of reproducing a fault from an FPGA testbench in simulation.

Verinject generated correct fault injection code for two larger processor designs – an 8051-compatible core [TS01] and a RISC-V core from the CArD course labs. Running the experiments on the RISC-V core led to a predicted dynamic cross-section of a matrix multiplication program to be 23.1% when the register file was targetted, compared to 46.71% for ground radiation testing of an 8051 processor [Rez01]. The difference can be explained by the program not using all of the available 32 general purpose registers, compared to very few registers in

the 8051, where most operations are performed through a single accumulator register.

5.1 Main contributions – summary

- Designing and implementing an algorithm for uniquely addressing all memory components in a Verilog design based purely on source code analysis
- Creating an memory-efficient method of fault injection into Verilog designs without tying the solution to a particular simulation framework
- Making scripts for generating reproducible and controllable inputs for the fault injector
- Designing and implementing a memory-mapped FPGA interface for the fault injector, based on an open-source AXI bus slave example
- Designing a simple “array adder” circuit and predicting its behaviour under fault injection with the use of mathematical techniques
- Comparing the predictions with the results from running Verinject on the circuit, in simulation and on an FPGA
- Evaluating the cost (in terms of performance and hardware) of fault injection based on the experiment
- Generating correct Verilog with fault injection for an Intel 8051-compatible design
- Generating correct Verilog with fault injection for a RISC-V code used in the Computer Architecture and Design course
- Measuring the dynamic cross-section of a matrix multiplication program running on the RISC-V processor, and comparing results with literature
- Producing a source code transformation-based fault injection tool usable on small and large Verilog designs

5.2 Future work

This project could be taken further by testing more Verilog designs and comparing the results with ones from ion beam experiments performed on equivalent physical hardware. Not all Verilog features are supported too, for example `generate` statements do not work in Verinject at all due to dynamic code generation, but could conceivably be supported.

Another possible interesting research avenue would be automatic fault finding and reporting. Currently this burden is placed on the testbench writer. There are techniques in the soft-

ware world such as fuzzing that allow controlled randomization of inputs based on static and dynamic code analysis. Applying these techniques to hardware with fault injection would allow for an even higher level of automation for testing, and this project could provide a basis for such a project.

This project could also be used to verify a design of a radiation-resilient processor, for example for a CubeSat satellite project or other relatively low-budget space applications. This was the original goal of this project in the first couple of weeks, but then the project group decided to split into smaller, individual project, so this goal was never attempted.

Chapter 6

Bibliography

[06] 'IEEE Standard for Verilog Hardware Description Language'. In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (Apr. 2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[20] *Rust Programming Language website*. 2020. URL: https://www.rust-lang.org/ (visited on 16/01/2020).

[Arm20] Arm. *AMBA – Advanced Microcontroller Bus Architecture*. 2020. URL: https://developer.arm.com/architectures/system-architectures/amba (visited on 06/03/2020).

[Eva+17] A. Evans et al. 'Heavy-Ion Micro Beam and Simulation Study of a Flash-Based FPGA Microcontroller Implementation'. In: *IEEE Transactions on Nuclear Science* 64.1 (Jan. 2017), pp. 504–511. ISSN: 1558-1578. DOI: 10.1109/TNS.2016.2633401.

[Gis20] Dan Gisselquist. *WB2AXIPSP: bus bridges and other odds and ends*. 2020. URL: https://github.com/ZipCPU/wb2axip/blob/master/rtl/demoaxi.v (visited on 06/03/2020).

[Joh17] Ian Johnston. 'Cosmic particles can change elections and cause planes to fall through the sky, scientists warn'. In: *The Independent* (Feb. 2017). URL: https://www.independent.co.uk/news/science/subatomic-particles-cosmic-rays-computers-change-elections-planes-autopilot-a7584616.html (visited on 30/10/2019).

[MV13] Wassim Mansour and Raoul Velazco. 'An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs'. eng. In: *IEEE Transactions on Nuclear Science* 60.4 (2013), pp. 2728–2733. ISSN: 0018-9499. DOI: 10.1109/TNS.2013.2267097.

[Rez01] S. Rezgui. 'Prédiction du taux d'erreurs d'architectures digitales : une méthode et des résultats expérimentaux'. In: (Jan. 2001). URL: https://tel.archives-ouvertes.fr/file/index/docid/163484/filename/PTE_118.pdf (visited on 06/03/2020).

[SBB19] Anderson Luiz Sartor, Pedro Henrique Exenberger Becker and Antonio C. S. Beck. 'A fast and accurate hybrid fault injection platform for transient and permanent faults'. In: *Design Automation for Embedded Systems* 23.1-2 (2019), pp. 3–19. DOI: 10.1007/s10617-018-9217-0.

[SM08] M. Shokrolah-Shirazi and S. G. Miremadi. 'FPGA-Based Fault Injection into Synthesizable Verilog HDL Models'. In: *2008 Second International Conference on Secure System Integration and Reliability Improvement*. July 2008, pp. 143–149. DOI: 10.1109/SSIRI.2008.47.

[Sny20] Wilson Snyder. *Verilator website*. 2020. URL: https://www.veripool.org/wiki/verilator (visited on 16/01/2020).

[TS01] Simon Teran and Jaka Simsic. *OpenCores 8051-compatible core*. 2001. URL: https://opencores.org/projects/8051 (visited on 06/03/2020).

[VRE00] R. Velazco, S. Rezgui and R. Ecoffet. 'Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection'. In: *IEEE Transactions on Nuclear Science* 47.6 (Dec. 2000), pp. 2405–2411. ISSN: 1558-1578. DOI: 10.1109/23.903784.