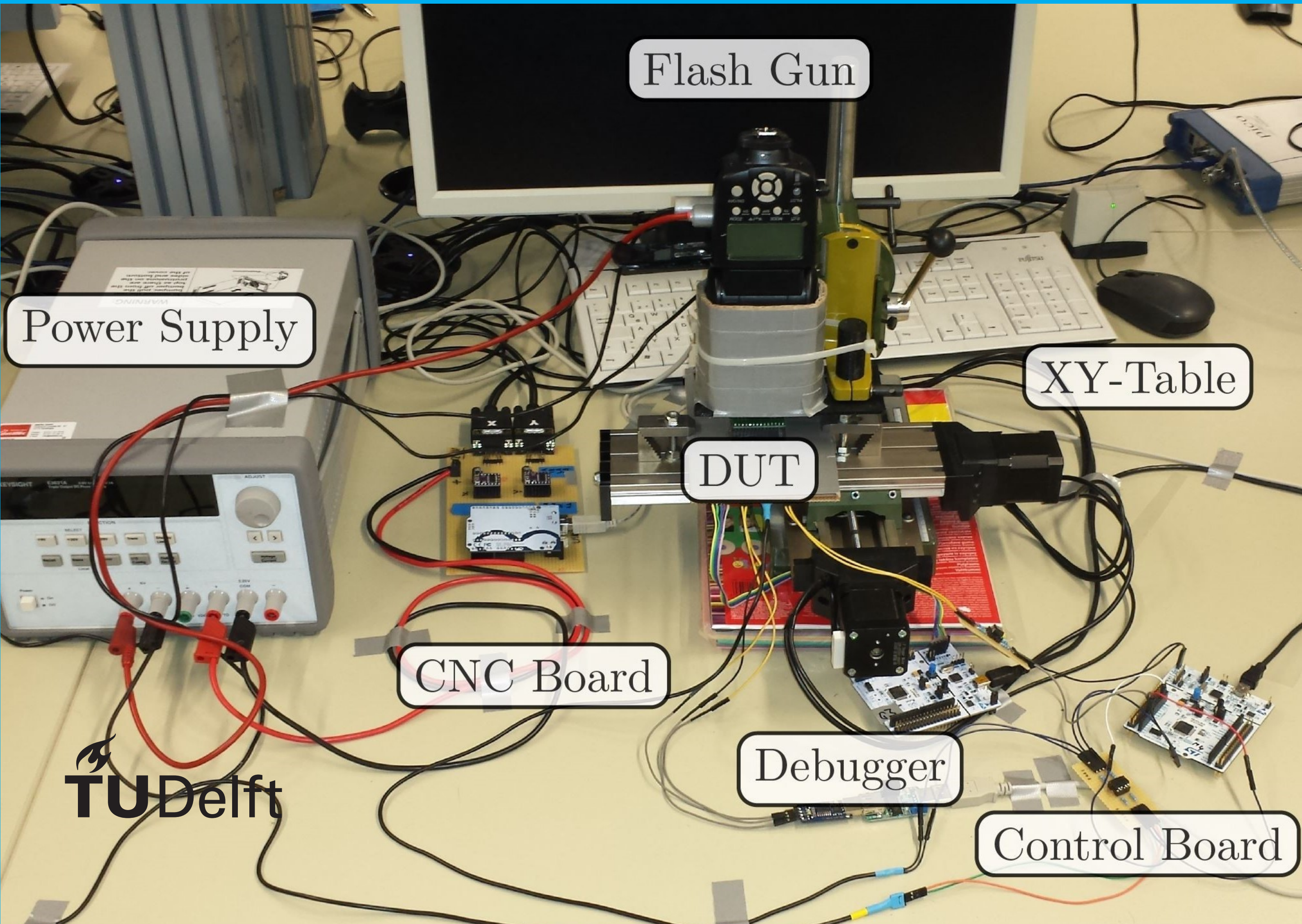
Fault Classification and Vulnerability Analysis of Microprocessors

Pavan Talluri



TU Delft

Fault Classification and Vulnerability Analysis of Microprocessors

by

Pavan Talluri

to obtain the degree of Master of Science
at the Delft University of Technology,

Student number:	4942256	
Project duration:	November 5, 2019 – November 4, 2020	
Thesis committee:	Dr. Ir. M. Taouil	TU Delft, supervisor
	Prof. Dr. Ir. S. Hamdioui	TU Delft
	Dr. Ir. R. van Leuken	TU Delft
	Ir. D. Vermoen	Riscure B.V.

TU Delft

Abstract

The adoption of Microprocessors is increasingly diversifying to several embedded and mobile devices. Growingly they can also be found in Smart Cards, RFID tags, SIM cards, Pay TVs, identity cards and passports. These devices store, processes and transact sensitive information like social security numbers and credit card numbers. Ensuring security of such systems is of paramount importance. Attackers use Fault injection as one of the techniques to induce faults into the processor in order to gain access to the sensitive information to abuse it. Vulnerability Analysis of the processors can help chip designers to counteract some of these risks. This analysis can be achieved by investigating the resulting fault space upon exhaustive simulation of fault injection attacks. Therefore, efficient tools and frameworks are needed to provide such security verification, where critical vulnerabilities can be discover and mitigated at design-time.

Multiple tools and frameworks for simulation based fault injection of hardware designs are available in literature, each with their own shortcomings. Two main strategies have been proposed in literature, one based on tool manipulation while other based on code modification. Applying tool manipulation, a designer can automate the process to inject faults in the system and obtain simulation results in a fast manner. However, this approach is limited by the features provided by the tool, which makes difficult to explore many different fault models. Additionally, in most cases, results require manual inspection to be interpreted. The latter approach, which is code modification, can inserts agents to cause the faults (i.e., saboteurs) or vary the existing design (i.e., mutants). It allows designers to achieve a high degree of control in terms of the type of fault and the injection method. However, current solutions are limited to a specific language, design or scenario. Hence, the literature presents many different strategies and tools to apply faults to investigate hardware behavior, but still the interpretation of vulnerabilities related to processors are not considered in such tools. Moreover, a complete automated framework capable to get a design and process it to report vulnerabilities and behavior issues related to security is still needed.

This thesis proposes a framework that provides a complete toolset able to evaluate vulnerabilities of processors in hardware description language. Its main steps comprises design instrumentation, simulation based fault injection and automatic fault classification. RISC-V is chosen as the target architecture due to its open source nature and its increasing adoption by academia and industry. Code profiling was carried out on the frameworks to identify bottlenecks to performance. The results were used to optimise execution time of simulations using the framework. Performing Fault injection campaigns requires running simulations in the order of 100k, which requires systems with high computing power to complete them in reasonable time. Therefore, multiprocessor support was implemented in simulation framework, which could be enabled or disabled during the injection campaign. The framework was used to perform fault injection campaigns on PicoRV32 and DarkRISCV processors. A comparison between the processors is made based on their major failure signatures. Analysis of finding design constructs in the processors which cause the major failure signatures was carried out. The results from this vulnerability analysis are used to propose software and hardware countermeasures to make the design more robust against fault injection.

Acknowledgements

I would thank all the following people without whom this thesis would have been far from completion. In no apparent order:

- My daily supervisor Dr. Ir. Mottaqiallah Taouil from TU Delft (NL) for all the guidance and support during the project as well as writing the thesis. I would like to thank Prof. Dr. Ir. Said Hamdioui, my supervising professor for his intriguing lectures and introducing me to the field of hardware reliability and security.
- Dennis Vermoen and Stefan Droege, supervisors at Riscure BV. Their insights and feedback were critical for what I have achieved in my thesis. Despite being busy, they always found time to answer all my questions and have weekly meetings to help me progress throughout the length of my thesis.
- Cezar Wedig Reinbrecht, for his suggestions and assistance in all the weekly meetings.
- My parents, Sankar and Kamala for their motivation and support, whom I owe everything to.

Pavan Talluri
Delft, October 2020

Contents

1 Introduction

1.1. Motivation

Since the invention of the transistor, silicon industry has made significant progress in improving the computational capacity, power and size of silicon devices by orders of magnitude. This complexity opened up several avenues for adversaries to develop new forms of hardware attacks [21]. One important target are the processors, which are increasingly seen in smart devices such as ID cards, SIM cards and smart cards [28]. This claim can be substantiated by the trend of increasing shipments of secure devices in Europe from 2010-2019 shown in Figure 1.1. These devices store sensitive information and are often targeted by attackers, who try to retrieve this information by compromising their security. One of the most dangerous threat are the fault injection attacks. The first article to discuss about fault attacks in literature is from 1997, by Boneh [16]. It lists the methodology followed in corrupting a computation by manipulating the circuit environment through fault injection. Since then, several new techniques have been designed to compromise or steal sensitive information causing great impact on silicon industry [35][62][15].

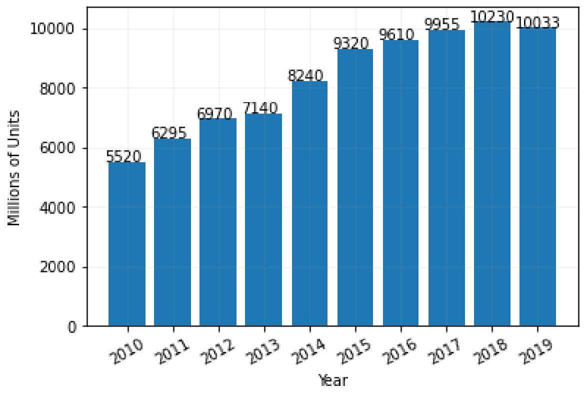


Figure 1.1: Secure Devices Shipments in Europe [4]

Fault injection can be achieved by various techniques such as variations in supply voltage, external clock, temperature, exposing to high intensity laser, electromagnetic radiation etc.[28]. Protecting systems against fault injection is quite challenging. Successful fault attacks were carried against DES implementation on a smart card [32], protected implementation of AES [72] as well as RSA authentication[10]. This is the reason why system designers require an abstraction of set of faults that could possibly be used by the attackers. Consequently, such knowledge allows engineers to build more effective software and hardware countermeasures [56]. This grouping of possible set of faults can be termed as a fault model. Most countermeasures are based on based on redundancy and this add considerable area to the design or increases execution time. This is due to the fact that, there exist no methodology to evaluate vulnerable parts of the design in the event of a fault. Therefore, to achieve an efficient methodology to explore the vulnerable parts of a design, a technique to stimulate the design under different fault models is needed.

Fault injection(FI) techniques can be classified into Hardware based, Software based, Simulation based, Emulation based and Hybrid [29]. Hardware based fault injection requires specialised equipment to inject faults. The focus of this thesis is to inject faults at HDL level and thus simulation based FI is adopted. Existing FI techniques focus on modifying the HDL code manually which is not desirable. This thesis overcomes this limitation by proposing an automated frameworks to instrument the design, inject faults and perform automatic fault classification. Vulnerability analysis of the processors as well as software and hardware countermeasures are discussed towards the end.

1.2. State of the Art

There exist multiple techniques of simulation based fault injection for any HDL like Verilog. Different classes of techniques are shown in the Figure 1.2. Saboteur is a special module added to the design, which if activated, alters the value or timing characteristics of a signal in the original design at the port level. It don't effect the design when inactive. A mutant is module that replaces the original design module, which when activated behaves like a module in presence of faults. Either behavioural descriptions of the design are modified or sub-modules are replaced with different modules. Simulator commands can also be used to modify values of either the signals or registers in a design. Unlike saboteurs and mutants, simulator commands doesn't require code modifications in the design. Simulation events to inject faults can also be invoked by using Verilog programming interface(VPI). VPIs are simulator independent and can be used to inject fault in any event region of a Verilog simulator.

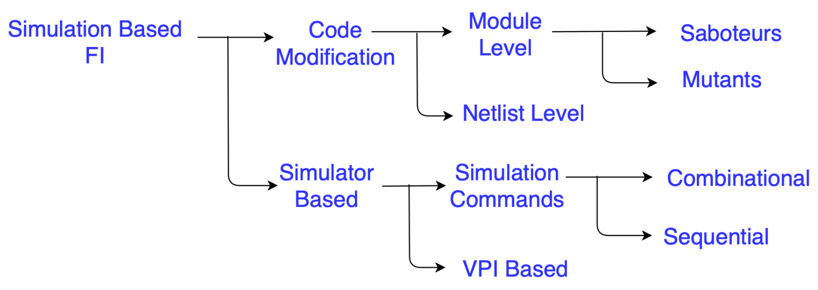


Figure 1.2: HDL based fault injection techniques [37]

Code Modification - Saboteurs: A tool for automatic placement of saboteur was described in [43]. This tool, unlike the previous ones, was built considering FI for emulating external attacks and thus included different fault effects that could be implemented by a saboteur such as stuck at 0/1, indetermination, delayed input etc. Though placement of saboteur is automatic, a port selection for the placement is to be manually selected from a provided GUI. This approach becomes unmanageable for complex designs. The tool does not do signal extraction from design, making choosing critical signals hard.

Code Modification - Mutants: An automated framework for mutant based fault injection approach was discussed in [44]. Multiple mutations have been considered by perturbing the model at conditions such as if and case statements, assignments as well as operators. These translate into Stuck-Then, Stuck-Else, Micro-operation, global stuck data, local stuck data fault models. Using the configuration mechanism, multiple models are generated with a single fault in their component when compared to the reference model. A static mechanism would allow only permanent faults to be injected. A dynamic approach was developed by using guard signals along with configuration mechanism to allow for injection of transient faults. In this approach, initially a fault free model is simulated till the time of fault injection. Simulator commands are used to stop the simulation and save the state to fault. The saved state is loaded to a model with a mutant and simulation is resumed. The drawback of this approach is its simulation time and memory required by simulator to store all the models. The simulation time was measured to be 100 times slower compared to the traditional simulation command based approach.

Code Modification - Netlist Level: The paper [12] discusses about a net list level fault injection tool for FPGAs. The framework uses a fault injection unit (FIU) hardware module in the FPGA to inject faults in RTL. The fault models modelled were stuck at 0/1, Random fault, Delay fault and SEU fault. The nets in the net list, which need to be injected with faults are opened and connected to FIUs. FIUs either drive the original value or a modified value to reflect one of the fault models mentioned earlier. Random faults are made possible due to the implementation of LFSRs. The timing and the type of fault model are controlled by a fault injection controller (FIC), which in-turn is programmed by the user. The user enter all the nets of the synthesized design to be subjected to fault injection. Random faults can also be controlled by the user by LFSR load value. A fault pattern file can also be used in-order to inject longer sequences of faults. The drawback of this solution are that, it requires a third party synthesis tool as well as signals present in HDL may not be present in netlist due to logic optimisation. The framework also doesn't provide a simulation script to generate a specific fault case in logical simulation.

Simulator commands - Combinational: Simulation based fault injection was earlier used to access the dependability of fault tolerance of systems towards radiation such as cosmic rays, known as single upset events[61]. This 1993 paper [50] proposes a VHDL based simulation tool, that can inject stuck at faults for simple designs. It fails to discuss about the scalability of the tool to more complex designs such as a processor and is language specific to VHDL.

Simulator commands - Sequential: A different approach entails fault injection into the model at the time of simulation by use of simulator commands. Here, the values of variables and signals of the model are varied by the simulator to model a fault. The paper [24] proposes a tool controlling the simulator to facilitate FI. Macros are used to prevent re-writing command code for every single fault. A precise control of fault duration, fault instant, fault value and fault signal/variable by the tool is possible. The drawbacks of this approach are, the fault simulation is very slow and cannot be used for fault space exploration; furthermore, the tool is simulator specific.

VPI: A Verilog VPI based approach was suggested in [18], which is independent of the simulator used unlike the simulator command based approach. Simulator command approach to inject faults into a non blocking statement is not possible as NBA(Non Blocking Assignment) is done in event region 3 of Verilog simulator as opposed to region 1. As modifying Verilog design code like in mutants and saboteurs is not required, recompilation for different fault injection simulations is not a necessity. Bit flip, stuck at 0/1, indetermination and high impedance were the fault models targeted. The developed framework reads inputs from user in an XML format, where the desired fault properties can be specified. ICORE processor was used to carry out the case study. The only shortcomings of the approach are its inability to scale to different type of faults as well as its performance for large scale fault exploration. This approach involves halting the simulator at the fault injection instant, injecting the fault, run simulation for specified fault injection period, halt the simulator, clear the fault and resuming the simulator again. The overhead of start-stop approach of the simulator as well as the need to manually add the required fault for the simulator hinder its use for a large scale fault space exploration.

1.3. Thesis Contributions

The main goal of this thesis is to design a framework to analyse how vulnerable is a processor to fault injection attacks. To achieve this objective, this thesis perform fault space exploration by injecting faults into the design, propose a classification for the faults and finally evaluate the methodology efficacy by embedding some countermeasures into a processor. The proposed framework is validated through two use cases; the RISC-V architectures namely PicoRV32 and DarkRISCV. As a result, we compare and contrast for vulnerabilities in their implementations. The primary contributions of the thesis have been the following.

- **Creation of a framework for automatic instrumentation of Verilog-2001 based design to facilitate fault injection**
  The framework employs, Pyverilog to convert the design described in Verilog-2001 to an abstract syntax tree (AST). This AST is read by a custom parser written in python, that instruments the code with fault injection signals. Finally, the framework utilises an AST code generator tool to convert the modified AST back to Verilog. All the registers and wires in the design can be injected with faults from the injection bus provided on the top level module of the design.
- **Development of a framework for simulation of designs which supports injection of fault in run time with automatic fault classification**
  The python framework converts the verilator based test bench to a C++ object, that is loaded as a run time library. The functions in the library are utilised by the framework to inject faults in the design when the simulation starts. The names of signals and registers where the fault are to be injected is provided as an input to the framework. Multiple simulations are run, each with a single fault in one of the bits of the aforementioned signals and registers. The framework also categorises the failed simulations to one of the fault categories. The framework has multiprocessor support, enabling it to be scaled to larger designs.
- **Exploration of the fault space on RISC-V based processors**
  An exhaustive search of fault space was performed with a single fault per simulation for RISC-V based processors. All the faults were grouped into multiple fault classes based on their resulting effect on the processor state.
- **Comparison of Fault space for different RISC-V based processors**
  A complete fault space exploration was performed for PicoRV32 and DarkRISCV, both of which implemented the RISC-V ISA. Analysis, comparing both the faults spaces illustrates the effect of different implementations of same ISA on resulting faults.
- **Proposing countermeasures for faults in the resulting fault space**
  Countermeasures were proposed for faults with highest frequency as seen in the simulations for PicoRV32. A watchdog timer was implemented in the design as a hardware countermeasure for the class of abrupt halt faults. A software countermeasure was envisaged to protect against control flow errors like 'branch not taken' class of faults.

1.4. Report Outline

The remainder of the thesis is divided into five chapters. Chapter 2 gives an introduction to different faults, fault models and fault injection attacks. These basics help to create a background to help dive into the later chapters. Chapter 3 discusses about the RISC-V architecture, ISA, available tool chains and open source cores. Fault injection corrupts the state of the processor and thus is checked on every clock cycle by the simulation framework. To understand what constitutes as a processor state, knowledge regarding the RISC-V architecture is of paramount importance. Chapter 4 presents the tool flow and framework that was used in this research, to instrument the design, inject faults and classify the faults automatically. Several tools like verilator, Pyverilog and Yosys that were integrated into the framework are also briefly discussed. Results of fault injection for PicoRV32 and DarkRISCV along with the implemented hardware and software countermeasures are examined in Chapter 5. Chapter 6 concludes the thesis as well as discusses about the future scope of this research direction.

2 An Overview of Faults and Fault Injection Attacks

This chapter introduces the concept of faults and its sources, fault categories, fault models, fault injection techniques and finally some fault injection frameworks.

2.1. Faults

A fault is an unpermitted deviation of atleast one characteristic property (feature) of the system from acceptable, usual, standard condition [36]. It is an abnormal condition that may cause reduction in, or loss of, the capability of a functional unit to perform a required function [31]. In the following subsections, we describe in details the sources of faults in a hardware system, how faults can be categorized and what are software/ISA faults.

2.1.1. Sources of Faults

Different faults are possible based on the type of source. They are shown in the Figure 2.1 and are discussed in more detail in the following sections.

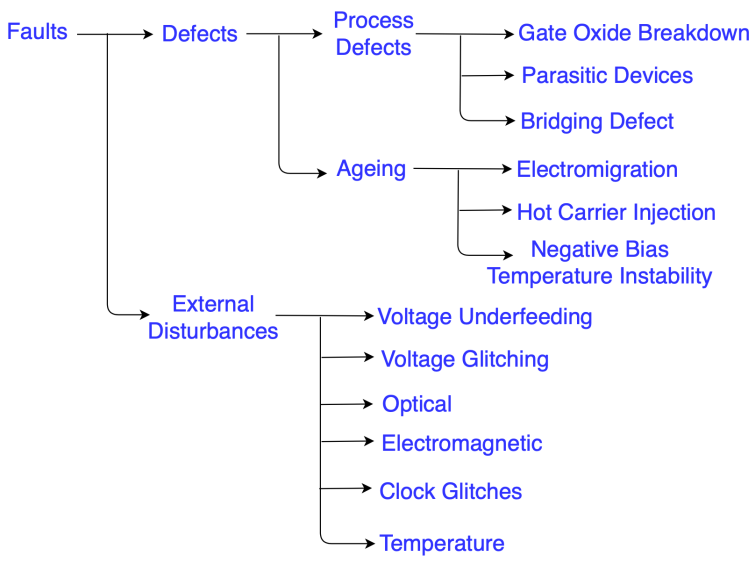
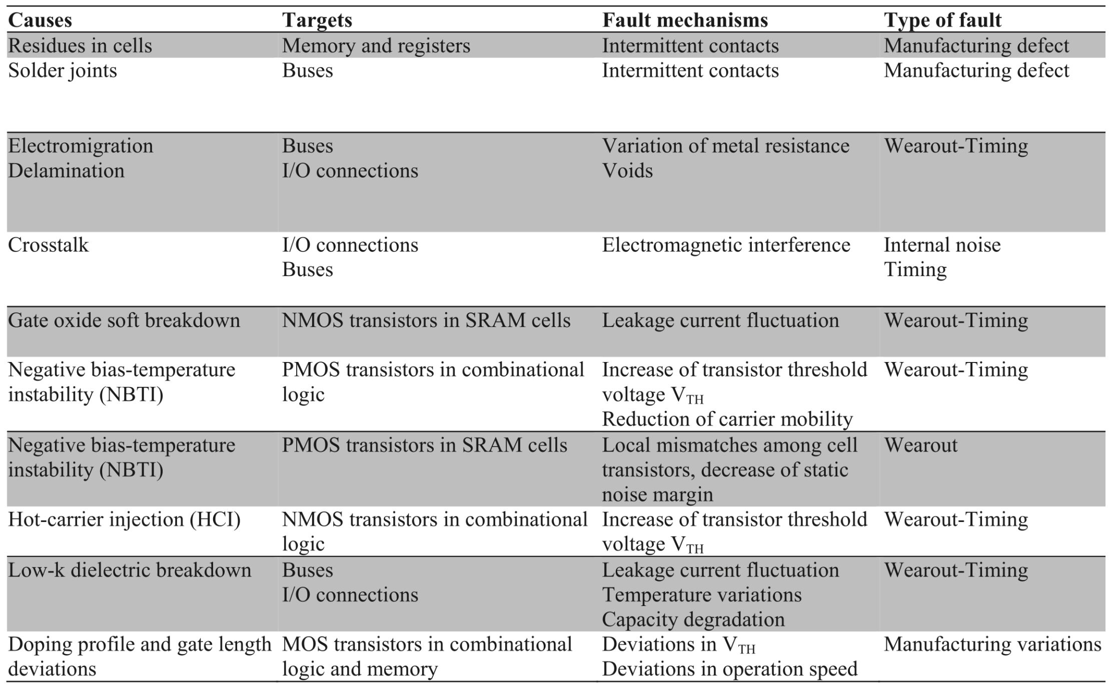


Figure 2.1: Types of faults

Defects

Manufacturing processes may induce imperfections in the intended CMOS structures, affecting their expected behaviour. These flaws are termed as process defects. In the course of their lifetime, ageing can also induce flaws in the CMOS based designs, termed as time dependent defects. Some defects are listed in the Figure 2.2. Environmental factors such as cosmic rays and radioactive particles can also induce faults into chips. Physical disturbances such as voltage underfeeding, lasers, EM pulses and clock glitching can be used by an attacker to induce faults in the chip in order to derive secret information or compromise its security. These are discussed in detail in the coming sections.



Causes	Targets	Fault mechanisms	Type of fault
Residues in cells	Memory and registers	Intermittent contacts	Manufacturing defect
Solder joints	Buses	Intermittent contacts	Manufacturing defect
Electromigration Delamination	Buses I/O connections	Variation of metal resistance Voids	Wearout-Timing
Crosstalk	I/O connections Buses	Electromagnetic interference	Internal noise Timing
Gate oxide soft breakdown	NMOS transistors in SRAM cells	Leakage current fluctuation	Wearout-Timing
Negative bias-temperature instability (NBTI)	PMOS transistors in combinational logic	Increase of transistor threshold voltage V_{TH} Reduction of carrier mobility	Wearout-Timing
Negative bias-temperature instability (NBTI)	PMOS transistors in SRAM cells	Local mismatches among cell transistors, decrease of static noise margin	Wearout
Hot-carrier injection (HCI)	NMOS transistors in combinational logic	Increase of transistor threshold voltage V_{TH}	Wearout-Timing
Low-k dielectric breakdown	Buses I/O connections	Leakage current fluctuation Temperature variations Capacity degradation	Wearout-Timing
Doping profile and gate length deviations	MOS transistors in combinational logic and memory	Deviations in V_{TH} Deviations in operation speed	Manufacturing variations

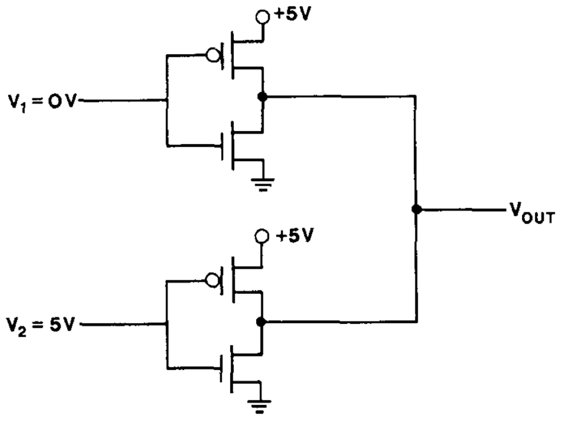
Figure 2.2: Defects and their fault mechanisms [17]

Process Defects

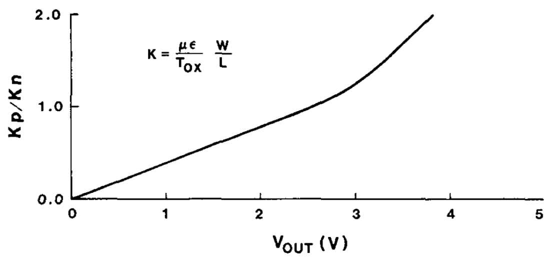
- **Gate Oxide Breakdown**
  Gate is more vulnerable to breakdown if the manufactured gate oxide is thin. When the transistor operates and the current flows though the gate, manufactured traps get charged and start damaging the oxide. This causes thermal damage to the transistor, which increases conduction through the traps and thereby creating more traps. A conductive path between the substrate layer and metal layer of the transistor is established. If the resulting effect is only higher leakage current, this phenomenon is called soft gate oxide breakdown. A manifestation of a cross section in the gate connecting the metal and substrate layer is termed as a hard gate oxide breakdown [47].

- **Parasitic Devices**
  Unintended structures like PN junctions, Bipolar/ MOS transistors can manifest in integrated circuits (IC) due to fabrication process variations and Design/layout geometric constraints. These parasitic devices can sometimes benefit the parametric performance of design, but a conscious effort is made to keep their effect low to maintain the functional intent of the IC. A decrease in output current drive, timing constraint violations, increased I_{DDQ} and I/O leakages can be attributed to parasitic devices in CMOS ICs. In a p-channel transistor, due to vertical and horizontal parasitic npn transistors in p-wells causes negative charge instability in undoped dielectric layer. For designs that should be radiation hardened, gate and field oxide transistors are particularly to be avoided. If design effort is not made to avoid these parasitic transistors, they would cause increased propagation delay, altering of logical functional intent of the IC and decreased maximum operatable clock frequency [70].

- **Bridging Defect**
  If a low resistance path occurs between logical nodes while wafer fabrication, it can introduce feedback or non-feedback configurations. A feedback can result in a latch or an oscillator, if the number of logical inversions are odd. A test set that guarantees 100% stuck at fault coverage doesn't guarantee absence of 2 bridges faults. A non-feedback bridging fault fuses outputs of two different gates. This defect can only be identified if both the gates are driven to opposite logic states. The relative transistor conduction constants of the pull-up and pull-down networks, which determine the current drive strength, determine the resultant voltage at the bridged node. The following Figure illustrates the ratio of transistor conductance constants of the two bridged CMOS inverters versus their output voltage at the bridged node [70].



(a) Bridged CMOS



(b) V_{Out} vs ratio of conduction constants k_p/k_n

Figure 2.3: Bridging Defect

Time Dependent (Ageing)

- **Electromigration**
  Due to the geometry of the metallic wires and the applied high electric field electrons migrate, causing voids and variations in the resistance. The voids can manifest as an open circuit and the accumulation can cause short circuits with adjacent wires. This completely alters the function of the circuit from intended behaviour. This is widely observed in Cu-Capping layer interface as it conducts the highest mass transport. Cu surface cladding can be deployed to minimize the effect. Scaling has further escalated the problem due to increase of Cu interface area in comparison to the volume. Degradation of wire due to EM depends on their geometry and current density. Unidirectional buses don't suffer from EM as they charge and discharge through the same end. Power, ground, clock and bidirectional data lines are all effected by EM [5].

- **Hot Carrier Injection**
  Scaling has reduced the channel length in the transistor, increasing its frequency of operation. This subsequently has led to higher electric field in the channel. Coupled along with higher voltages, this can give electrons or holes enough kinetic energy to move from substrate to gate oxide. The mechanism involved could either be impact ionisation or scattering that causes this interface state generation. The threshold of the transistor is in turn effected by this spurious injected current. Reduction of V_{DD} (operating voltage) along with LDD implants help to mitigate Hot carrier injection to an extent. Below the gate length of 50nm, the gate voltage was not reduced enough to compensate for scaling, to keep the HCI low. Emphasis on HCI will continue to increase as transistors are scaled further [39].

- **Negative Bias Temperature Instability**
  Operating voltages of devices has been reducing considerably over the decades. The electric field in the gate and operating temperature on the other hand have gone up due to higher power dissipation and lower voltage difference between V_T and V_{Gate}. These higher operating temperatures and low gate oxide thickness cause silicon hydrogen bonds in gate substrate interface to

break. These vacant silicon ions function as holes altering the threshold voltage of the transistor. Thus, transistors Inversion channel mobility decline, channel threshold voltage shift and subsequent slow down of the transistor causing timing faults over a period of time can be attributed on NBTI[52]. As a example, due to NBTI in a MOS capacitor, a positive flat band shift due to higher positive charges near Si-SiO_2 interface can be seen initially. Further, a negative flat band shift is obtained by exchange of charge with silicon substrate, increasing net positive charge in Si-SiO_2 interface [41]. These are shown in the Figure 2.4 illustrates this effect.

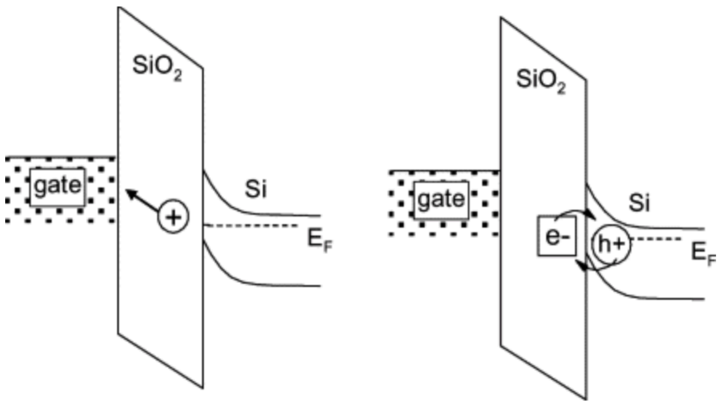


Figure 2.4: Effect of NBTI on MOS capacitors[41]

Physical/External disturbances

- **Voltage Underfeeding**
  Constant voltage underfeeding causes the rise in setup time of the combinational logic to attain a stable state, which inturn effects the maximum operable clock frequency of the circuit. If the frequency is still maintained as per the normal operating conditions, while decreasing the supply voltage, proper setup of slower logic paths will fail[7]. A precision power supply such as agilent 34420A, can be used in a testbench to control the power supply of device under attack. A similar workbench was described in [8] and is shown in the Figure 2.5. This paper describes that only **LOAD** instructions are effected by this approach. This can be attributed to the fact that, **LOAD** instructions have long data paths compared to **STORE** or other arithmetic instructions. Write back buffers shorten critical paths of **STORES** and arithmetic instructions use functional units, registers that are highly optimised. All the failing loads cause bit flip down errors and no stuck at '1' errors. The result is an instruction swapping error if the **LOAD** corresponds to an instruction fetch and a data load error if its a data access. The spacial precision of the faults is limited in this approach as power is distributed all over the chip by the power distribution network. The temporal precision is also low as voltage underfeeding effect persists over multiple clock cycles [11].

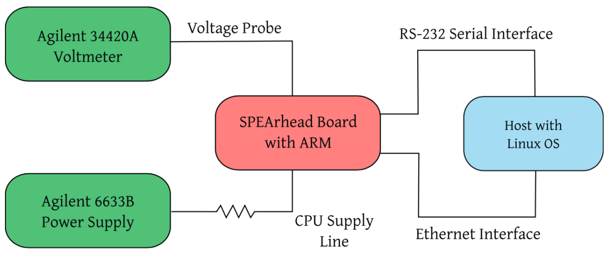


Figure 2.5: Voltage Underfeeding Workbench [8]

- **Voltage Glitching**
A temporary drop or spike in the supply voltage can cause timing violations, that can be used for an attack. Unlike voltage underfeeding, more control over the temporal location and intensity are possible in this approach. In the paper [27], voltage glitching was used to effect the bias of the generated output random numbers. Newer chips can have multiple power domains that can be effected by switching between multiple voltage levels. Faults can be injected into a particular power domain without effecting others. Attacks on controlling PC in an ARM 32 bit processor by this method was discussed in the paper [57]. A single or multiple bit corruption in the load instruction was induced to load an attacker desired value in the PC to allow execution for arbitrary code. If countermeasures are in place to prevent such attacks, the paper [45] describes a method to inject a transient timing fault by application of positive bias voltage on the substrate of MOS transistor. This application of few mV, changes the threshold voltage of transistor and introduces a local voltage pulse. This technique is referred to as Forward Body Biasing Injection (FBBI) and the platform used for this is described in the Figure 2.6.

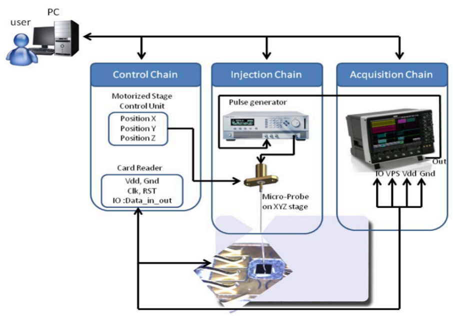


Figure 2.6: Forward Body Bias Injection Platform [45]

- **Optical**
This is the most precise and effective fault injection technique. The IC is first decapsulated from the packaging and is exposed to a light pulse. The source could be a low cost flash light or a laser beam. The latter achieves spatial precision in the order of micrometers and temporal precision in nanosecond range. Even a single transistor can be targeted by choosing the energy and duration of light pulse accordingly. On the flip side, the target chip could be permanently damaged. Shorter wavelengths are used to penetrate the metal layers if the front side of the chip is targeted. Infrared light is used to penetrate silicon substrate if backside is attacked. Multiglitching, where multiple faults are injected in a short period is also also possible. The state of the transistor, setting or clearing a single bit in memory can be manipulated, leading to faulty computations [21].

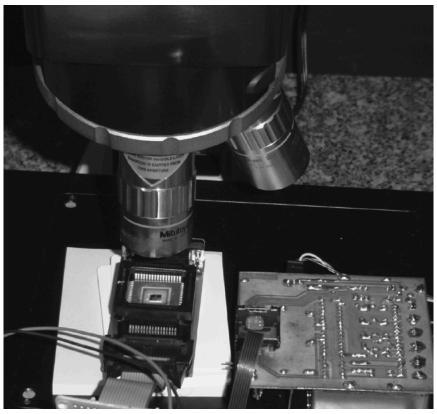
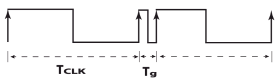


Figure 2.7: Laser Fault Injection [21]

- **Electromagnetic**
  By driving a high current through a coil, electromagnetic radiation as transient or harmonic pulses can be emitted, that can be used to inject faults in the target. Localised faults, that affect only a part of the chip are possible. Decapsulation of the chip as in optical faults is not required in EM attacks, though its spacial precision is comparatively lower. EM pulses induce eddy currents, which translate to faults. Injection probe's position and applied voltage determine the position and intensity of the generated EM pulse. Single bit faults in the memory can be induced by eddy currents on the surface of the chip. Errors in program flow and SRAM contents due to EM were discussed in the paper [68].

(2.8a) EM Fault Injection [68]

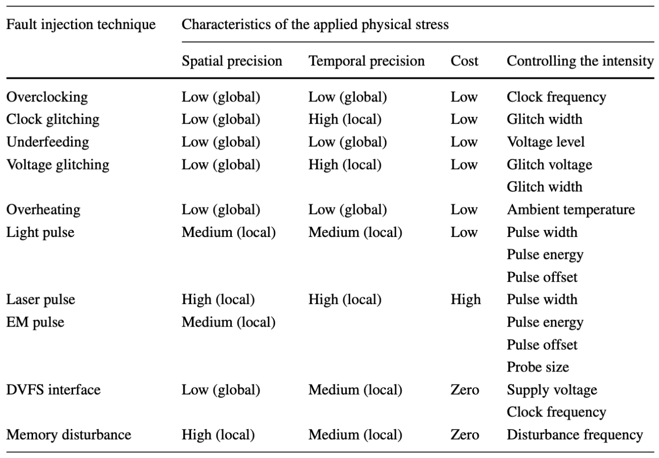


(2.8b) Clock Signal glitching [21]

- **Clock Glitches**
  If an external source supplies clock to the circuit, it can be switched between a faster and slower clock. Temporarily the width of a clock pulse could be shortened by an attacker in clock glitching. This causes setup time violations and incorrect values are latched by the logic paths in the circuit. The temporal precision is very high as the clock cycle at which this glitch is introduce as well as how long the glitch is introduced can be controlled. This can translate to premature commit of the current instruction executing in a processor or capturing wrong data in registers or memories [21].

- **Temperature**
  Exposure to too high or too low temperatures outside the chips operating temperature range can induce faults. Overheating can induce timing violations in data path or cause bits to flip in memory cells. A focus on particular targeted portion of data is not possible. Some NVMs (Non volatile memories) have different temperature thresholds for read and write. By maintaining the temperature so that only write work and not the reads, different attack scenarios can be mounted. A laser can be used to heat a particular area of the chip, giving a high spacial precision. A high intensity light bulb or an alcoholic cooler on the other hand will offer a very low spacial precision [65][33].

The Figure 2.9 lists out the various fault injection techniques and their different characteristics.



Fault injection technique	Characteristics of the applied physical stress			
	Spatial precision	Temporal precision	Cost	Controlling the intensity
Overclocking	Low (global)	Low (global)	Low	Clock frequency
Clock glitching	Low (global)	High (local)	Low	Glitch width
Underfeeding	Low (global)	Low (global)	Low	Voltage level
Voltage glitching	Low (global)	High (local)	Low	Glitch voltage
				Glitch width
Overheating	Low (global)	Low (global)	Low	Ambient temperature
Light pulse	Medium (local)	Medium (local)	Low	Pulse width
				Pulse energy
				Pulse offset
Laser pulse	High (local)	High (local)	High	Pulse width
EM pulse	Medium (local)			Pulse energy
				Pulse offset
				Probe size
DVFS interface	Low (global)	Medium (local)	Zero	Supply voltage
				Clock frequency
Memory disturbance	High (local)	Medium (local)	Zero	Disturbance frequency

Figure 2.9: Fault Injection Techniques and their characteristics [11]

2.1.2. Fault Categories

Multiple fault categories are possible based on their temporal characteristics. They are shown in the Figure 2.10. A detailed description of the faults is given in the remainder of the section.

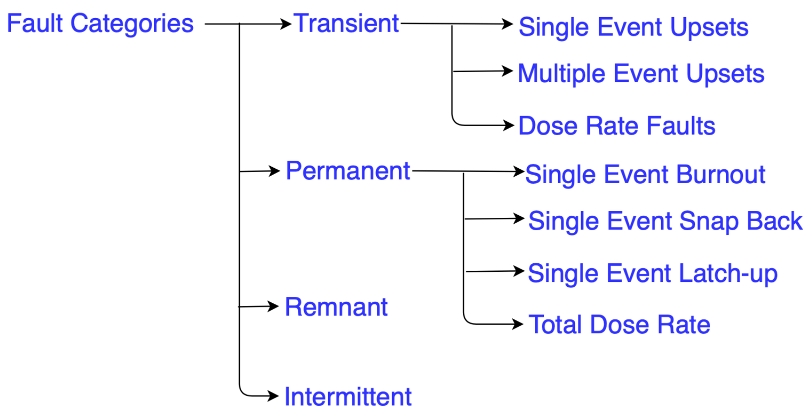


Figure 2.10: Fault Categories

Transient

These faults can be generated by environmental conditions like cosmic rays or fault attacks like EM pulses. Their effect normally lasts only for few clock cycles and wears off as soon as the fault source ceases to exist. The effect though could propagate to software level in a processor. Original intended behaviour could be restored by a system reset [28].

- **Single Event Upsets**
  A single bit either in memory or registers could be flipped to a complementary state. SEUs can also manifest as variations in power supply voltage or system clock.
- **Multiple Event Upsets**
  If multiple SEUs occur simultaneously in the system, it is termed as MEUs. High packaging density predisposes devices to MEUs.
- **Dose Rate faults**
  Particles whose individual effect is negligible, but whose aggregate effect translates to a fault are categorised as dose rate faults.

Permanent/Destructive

Permanent faults are due to manufacturing defects, wear-out mechanisms and fault attacks, where the physical defect cannot be reversed. A fault injection technique like laser pulse can also damage a register or memory. The effect of these faults persist indefinitely. Permanent faults in a processor are more likely to effect the software, causing a system failure. They have a less likelihood of getting masked and so it should be diagnosed and the system must repaired or reconfigured to avoid the faulty unit.

- **Single event burnout**
  This is due to thermal runaway in MOS transistors as a result of parasitic thyristors. The entire circuit could be permanently damaged due to this [23].
- **Single event snap-back**
  This fault is mostly observed in devices with high supply voltage. Self sustaining currents are induced in N-channel of MOS transistors as a result of parasitic bipolar transistors.

- **Single event latch-up**
  A parasitic PNPN bipolar transistor can create a self sustaining current in MOS devices, that can potentially damage the device permanently. This is illustrated in the Figure 2.11.

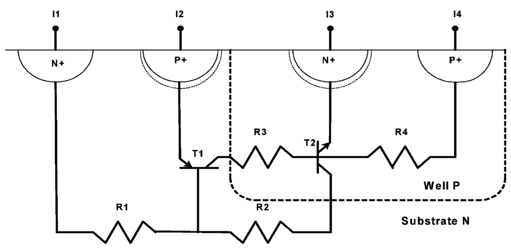


Figure 2.11: SELs - parasitic transistors - T1 & T2

- **Total dose rate**
  Exposure to environment can also damage the device and induce faults. Radiation can cause effect gate potential and degrade current characteristics. As a solution, the radiation hardness of the device can be improved [60].

Remnant

If the configuration memory in a SRAM based FPGA is injected with a fault, the architecture of the design is altered. FPGA should be reprogrammed to get rid of the fault [20].

Intermittent

These faults recur at the same location non-deterministically and are present for finite number of clock cycles. The architecture vulnerability due to intermittent faults depends mostly on the fault length. Though underlying hardware defects such as device wearout and manufacturing defects are their main cause, they are not present indefinitely. They cause stuck-at and bridging faults [47].

2.1.3. ISA/Software Faults

- **Instruction Replacement**
  When the instruction is in the pipeline of a processor and its binary encoding are corrupted during a bus transfer or when in pipeline registers, the instruction is transformed into an another instruction. This fault is called an instruction replacement fault [54]. If an assembly instruction can be skipped or if the instruction can be altered to not effect any useful register and as a result act as a **nop**, its is categorised as an instruction skip, special case of instruction replacement. These type of faults can be used by the attackers to bypass key checks in protocols like AES or RSA. Several other attack modes such as skipping a data load, backward jump, test inversion and changing a data load address are possible. The Figure 2.12 illustrates a C code and its corresponding assembly. It shows how if the jump instruction at line 9 can be replaced with a **nop**, the else condition check in the C program can be bypassed.

- **Device Reset**
  A complete reset of the device can be triggered by an attack like Forward Body biasing injection [45]. This completely corrupts the current state of the processor and the subroutines executed at reset or the boot loader can be exploited for an attack.

- **Data corruption**
  Multiple attack scenarios cause unique data corruption software faults. A clock glitch attack can cause the read of data bus even before the memory has the time to update the bus with the actual value, causing a data misread. This glitch attack particularly targets when the data is being transferred from memory to registers. When a laser is used to attack the data bus, it causes the read value to be always **255 (0xFF)** irrespective of the real read value. A voltage attack on **EEPROM** by increasing the supply voltage $\mathbf{V_{cc}}$ to the maximum operable circuit voltage, makes

```
char example(char u)
{
  char res, b;
  c = u + 5;
  b = c < 10;
  if (b){
   res = c + 1;
  }
  else {
   res = 0;
  }
  return res;
}
```

```
_example:
   ...
   mov r2,dpl   // load the parameter in r2
   mov a,#0x05 // put 5 into a
   add a,r2     // compute u + 5 in a
   mov _c,a     // store c into RAM from a
   clr c        // clear the carry
   subb a,#0x0A // computes b i.e. c−10
   jnc 00102$  // jumps to 102
                // if carry not set (else)
/* @ATTACK_ADDR1 */
   mov a,_c     // load c into a
   inc a        // a++ i.e. c + 1
   mov r2,a     // r2 stores a (res = c + 1)
   sjmp 00103$ // jump over else
00102$:
   mov r2,#0x00// r2 stores 0 (res = 0)
00103$:
   mov dpl,r2   // push r2 on the stack
   ret          // returns
```

Figure 2.12: C code and its corresponding 8051 assembly [58]

the data to be read as **zero** always [28]. Attacks to alter the memory directly in register or volatile/ non-volatile memory cause a single bit or multiple bits to either flip or to be stuck at 0/1 [75].

- **Computation error**
  When a Cryptographic chip, implementing AES is targeted by EM pulses, it can corrupt the computations and produce erroneous cipher texts. The circuit operation however is not halted. The paper [66] discusses this approach and cite an interesting effect of the type of the injector used. They point out that only crescent injector produces erroneous ciphers and not a flat head injector. The Figure 2.13 shows the areas on the chip that produce a computation error in cipher when attacked by EM pulses. A similar attack on smart cards can also impact the quality of the random numbers generated that are used internally[58].

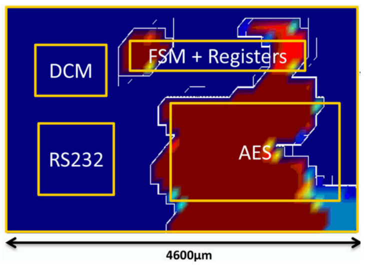


Figure 2.13: EM attack on a Cryptographic chip [66]

2.2. Fault Injection Techniques

Fault injection techniques were long used before fault attacks to affirm the dependability of a system. The device's behaviour is analysed when an unexpected event arises to ascertain if it is intended or not. To inject faults into a system prototype or model, various techniques have been designed and they fall into the following five categories [43]. The table 2.14 contrasts various techniques.

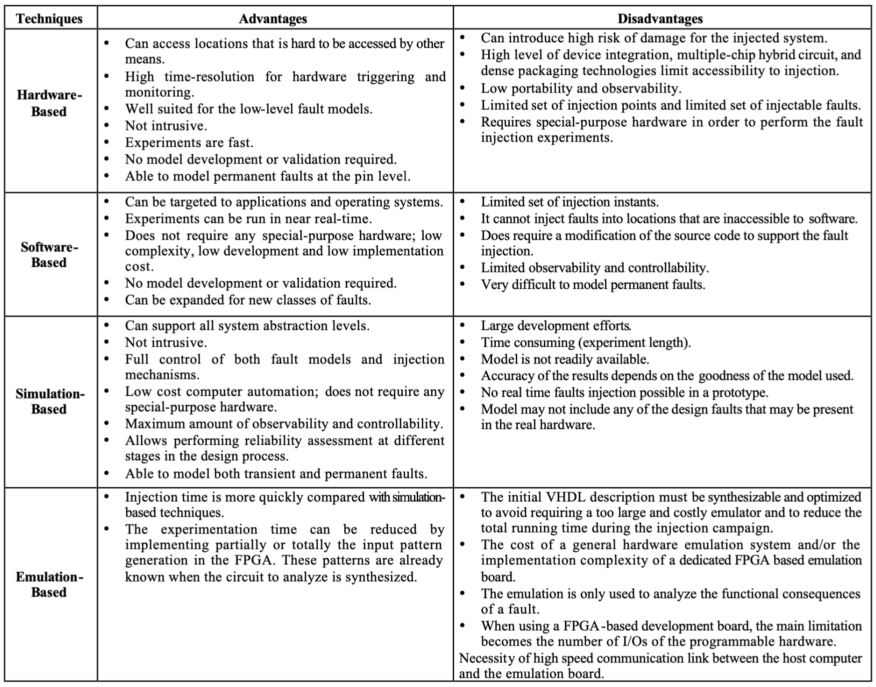
- **Physical/Hardware**
  The fault is injected directly at the physical level by disturbing the hardware parameters like voltage, clock frequency, temperature etc. This requires detailed knowledge, physical access to the device and may require specialised equipment to inject faults [44].

- **Simulation**
  This is the most frequently used method. The advantages of this method include flexibility as well as not requiring the physical device. Faults are injected at high level models like a Verilog based HDL model. On the flip side, the simulation time required to simulate all the faults is impractical. Various approaches such as simulation command based, saboteurs and mutants[29].

- **Emulation**
  To reduce time spent in fault simulation campaigns or to avoid expensive physical testing, FPGA based fault injection techniques were proposed. This requires the initial HDL model to be synthesisable, so that it can be used to reconfigure the FPGA. Actual behaviour of the device in the application environment with respect to real time interactions can be studied[44].

- **Software**
  If the errors that would have been produced upon the fault occurring in the hardware are reproduced at the software level, it is termed as software based fault injection[29].

- **Hybrid**
  A mix of software implementation and Hardware monitoring. Software techniques mask their presence and have no effect on the system other than the fault. Hardware techniques leave a footprint while fault injection. These techniques try to balance both approaches while maximising the effectiveness of fault injection campaigns[43].



Techniques	Advantages	Disadvantages
Hardware-Based	• Can access locations that is hard to be accessed by other means. • High time-resolution for hardware triggering and monitoring. • Well suited for the low-level fault models. • Not intrusive. • Experiments are fast. • No model development or validation required. • Able to model permanent faults at the pin level.	• Can introduce high risk of damage for the injected system. • High level of device integration, multiple-chip hybrid circuit, and dense packaging technologies limit accessibility to injection. • Low portability and observability. • Limited set of injection points and limited set of injectable faults. • Requires special-purpose hardware in order to perform the fault injection experiments.
Software-Based	• Can be targeted to applications and operating systems. • Experiments can be run in near real-time. • Does not require any special-purpose hardware; low complexity, low development and low implementation cost. • No model development or validation required. • Can be expanded for new classes of faults.	• Limited set of injection instants. • It cannot inject faults into locations that are inaccessible to software. • Does require a modification of the source code to support the fault injection. • Limited observability and controllability. • Very difficult to model permanent faults.
Simulation-Based	• Can support all system abstraction levels. • Not intrusive. • Full control of both fault models and injection mechanisms. • Low cost computer automation; does not require any special-purpose hardware. • Maximum amount of observability and controllability. • Allows performing reliability assessment at different stages in the design process. • Able to model both transient and permanent faults.	• Large development efforts. • Time consuming (experiment length). • Model is not readily available. • Accuracy of the results depends on the goodness of the model used. • No real time faults injection possible in a prototype. • Model may not include any of the design faults that may be present in the real hardware.
Emulation-Based	• Injection time is more quickly compared with simulation-based techniques. • The experimentation time can be reduced by implementing partially or totally the input pattern generation in the FPGA. These patterns are already known when the circuit to analyze is synthesized.	• The initial VHDL description must be synthesizable and optimized to avoid requiring a too large and costly emulator and to reduce the total running time during the injection campaign. • The cost of a general hardware emulation system and/or the implementation complexity of a dedicated FPGA based emulation board. • The emulation is only used to analyze the functional consequences of a fault. • When using a FPGA-based development board, the main limitation becomes the number of I/Os of the programmable hardware. Necessity of high speed communication link between the host computer and the emulation board.

Figure 2.14: Comparison of FI techniques [29]

2.3. Fault Models

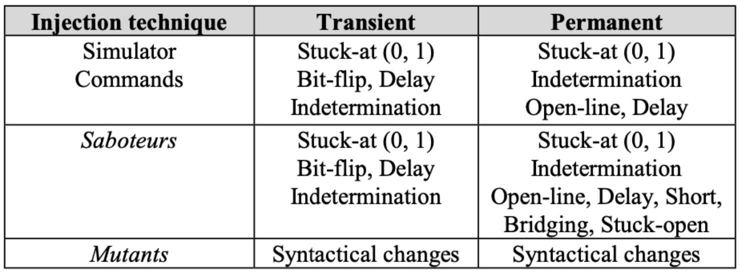
To model the complex behaviour of a fault and its effect taking the abstraction level into consideration, a fault model is defined. If the faults are modelled at a signal level, the following fault models are conceivable [44][53].

- **Stuck-at-0**: The signal value is forced to be ’0’ due to the fault.
- **Stuck-at-1**: Fault forces the signal to be always ’high’.
- **Stuck-open**: After the retention time, the signal is forced to be ’low’.
- **Bit-flip**: The effect of the fault is reading the signal value and flipping it.
- **Open-line**: If the fault causes an open connection, the signal value can be modelled as ’high impedance’ or **’Z’**.
- **Delay**: Signal is updated to its expected value after a delay.
- **Indetermination**: The signal is written with a random value **’X’**.

If the faults are modelled at the syntactical structure level of HDLs, they result in the following fault models[37].

- **Stuck-then**: An **If** condition is replaced by **True**.
- **Stuck-else**: An **If** condition is replaced by **False**.
- **Assignment control**: An assignment operation is disturbed.
- **Dead process**: A process is never made to execute by removing its sensitivity list.
- **Dead clause**: One of the clauses in a **Case** statement is removed.
- **Micro-operation**: An operator is disturbed by replacing it with an another operator.
- **Local stuck-data**: The value of either a signal or variable is forced to a value.
- **Global stuck-data**: All the updates to either a signal or variable are completely removed in the entire HDL description.

The following table summarizes various fault models based on different simulation based fault injection techniques.



Injection technique	Transient	Permanent
Simulator Commands	Stuck-at (0, 1) Bit-flip, Delay Indetermination	Stuck-at (0, 1) Indetermination Open-line, Delay
Saboteurs	Stuck-at (0, 1) Bit-flip, Delay Indetermination	Stuck-at (0, 1) Indetermination Open-line, Delay, Short, Bridging, Stuck-open
Mutants	Syntactical changes	Syntactical changes

Figure 2.15: Fault models for different simulation based Fault injection Techniques [37]

2.4. Fault Injection Frameworks

Multiple frameworks have been proposed for different fault injection techniques, the following are widely used and thoroughly described in literature.

- **Chiffre**
  This framework can be used to instrument the design automatically while also facilitating run time injection of faults. Chisel hardware description language should be used to code the design for this framework to work. This hardware construction language(HCL) enables early stage design emulation and security verification. A circuit compiler like FIRRTL is used to read HCL based design and emit it's equivalent Verilog code after optimisation and transformations to add fault injections as well as scan chains. The fault injection is done at module level and thus the amount of user effort required for instrumentation is very low. Both registers and wires in the design can be instrumented at module and submodule level, offering high scalability for larger designs.

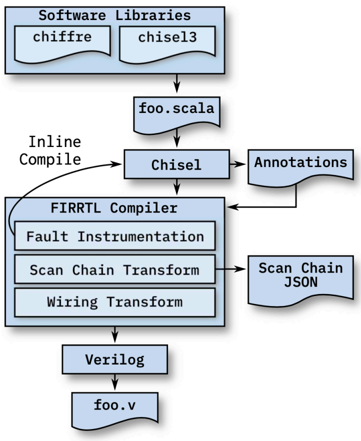


Figure 2.16: Chiffre : Chisel to Verilog generation and instrumentation [69]

- **Modelsim with scripting**
  Modelsim is a simulation and verification tool for digital designs. This approach involves simulation based fault injection using Modelsim and script to control the simulation, which in-turn control the flow of the program. A random point in the processor such as in a register, cache memory or flip-flop are targeted at a random time with a single or multiple bit flips. Different faults such as SEU (Single event upsets), SET (Single Event Transients), MET (Multiple Event Transients) and MBU (Multiple Bit Upsets) can be propagated into the design. Of the total run time, 5% to 80% can be used for the activation of the fault and the remaining time is normally used by the processor for warm-up and to execute diagnostic routines. The paper [30] uses this methodology for fault injection in LEON3 processor and found out that integer and multiplier units are more susceptible to single and multiple faults respectively.

- **Coppelia**
  Coppelia is an end-to-end tool to find security threats of hardware vulnerabilities. The tool reads the processor design file, security critical constraint file and finds several assert statement violations. Using backward symbolic execution, a path from reset to assert failure is traced. Clock stitching is used to handle symbolic execution over multiple clock cycles. Cone of influence and search optimisations are used to generate exact sequence of instructions that trigger the assertion fail. Coppelia thus automatically generates replayable exploits with sequence of inputs to trigger a bug. Coppelia is built upon KLEE, a popular symbolic execution engine. It also uses verilator to convert the HDL based design to cycle accurate C++ code. Tool flow of Coppelia is shown in the Figure 2.17.

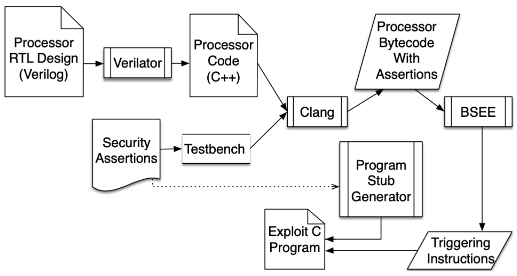
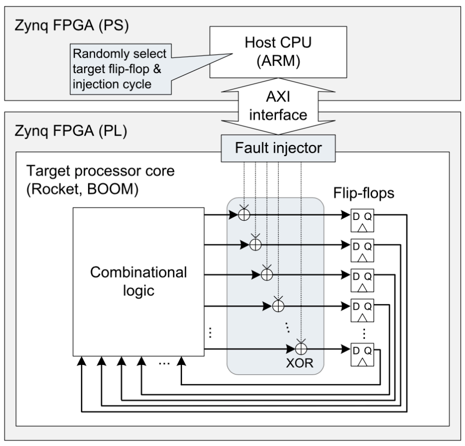


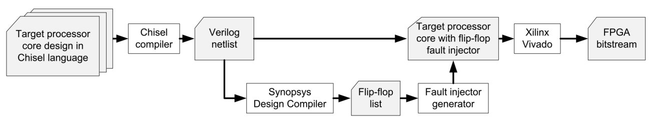
Figure 2.17: Workflow of Coppelia [64]

- **VerFI**
  A framework designed especially to inject faults into cryptographic hardware. VerFi is a simulation based fault injection tool that works on netlist of the design implementation. Fault injection can be acheived at Bit level granularity. The design could be inputted in the form of HDL or netlist, which is synthesised by either Synopsys or Yosys. The synthesizers are controlled by VerFI to not optimise redundancy, which is used as a fault countermeasure. The synthesizers also generate a configuration file, which the user could use for personalised fault injection. VerFI reads the fault configuration file along with input test vectors, simulates the design and produces coverage for the set of inputs as well as a report on the total number of generated faults. The faults are modelled in the circuit as gate level faults. Fault injection mechanism could be either hierarchical or component-wise. A event driven simulator is used, whose performance can be optimised by parallel fault simulation to use AVX2 on 256 bits, instead of bit-wise operations [75].

- **Emulation based flip-flop fault injection**
  Designs that are created using Chisel are compiled to generate a Verilog netlist. This netlist is synthesised using Synopsys Design compiler to produce a list of flip-flops in the design. For every flip-flop in the list, a **XOR** gate is added at the data input. One of the inputs of the **XOR** gate is connected to a fault injection signal, which can be driven to flip the bit to inject the fault. This instrumented design is then ported to the programmable logic side of an FPGA. The FPGA has a host CPU on the processing system, which can be clocked as high as 50MHz. All the fault injection signals of the design are controlled by the host CPU through an AXI interface. During the fault campaign, to inject a fault, host CPU drives one of the fault injection lines that is selected randomly at a random clock cycle. After the design finishes the execution, results are collected by host CPU to classify fault effect. Silenced data corruption, unexpected termination, hang and vanished are the outcomes that are selected as they are mutually exclusive [14].



(a) Processor Emulation on Xilinx Zynq FPGA



(b) Automated tool flow

Figure 2.18: FPGA emulation based Fault Injection Flow

2.5. Countermeasures

To thwart fault injection attacks on processors and applications running on them, countermeasures implemented both in software and hardware were proposed [40]. In-order to best protect the device, a combination of both software and hardware countermeasures are used. Hardware implementations are limited by manufacturing costs and software by performance. The goal of countermeasures is to make fault injection expensive to perform, but not to prevent them completely due to aforementioned trade offs.

2.5.1. Software Countermeasures

Software Countermeasures are a low cost approach as design changes in the processor are not required for their implementation, but rather a compromise in increase of execution time. They are also flexible as software can be changed to chose between various implementations. Following are some techniques and their brief description,

- **Functional level temporal redundancy**: Critical functions are called twice with same input data and their return data are compared for consistency [28]. This helps in fault detection, with a cost of doubling the execution time. If fault correction is desired, the function can instead be called thrice and their result be selected based on a voting logic [54]. In case of an encryption algorithm, running the decryption algorithm on the data produced and comparing the result with the input of encryption algorithm can detect faults.

- **Instruction Duplication/ Triplication** : The previous approach involves duplication at algorithmic level, where as this approach is at assembly instruction level. Same instruction is executed multiple times for error detection. The process can be automated by enhancing the compiler by adding compiler switches to implement the functionality [55]. Instruction duplication for a **load** to **r4** is shown in Figure 2.19a and Instruction Triplication of **xor** between **r1** and **r2** with result stored to **r4** is shown Figure 2.19b.

```
ldr r4,[r7];
ldr r12,[r7];
cmp r12,r4;
bne <error>;
```

(a) Instruction Duplication

```
eor r12,r12,r12;
eor r10,r1,r2;
eor r0,r1,r2;
eor r4,r1,r2;
cmp r4,r10;
eoreq r12,r12,#1;
cmp r4,r0;
eoreq r12,r12,#2;
cmp r10,r0;
eoreq r12,r12,#4;
cmp r12,#0
beq <error>;
cmp r12,#4;
moveq r4,r0;
```

(b) Instruction Triplication

Figure 2.19: Instruction Redundancy Techniques

- **Parity Checking**: Used for checking faults in data returned by load instructions. For data labelled as protected value, parity is pre-computed and stored in memory and a processor register is allocated to point to it. Later in program execution when a load instruction reads the protected value, its parity is computed and compared with pre-computed parity to determine its consistency. The drawback with this approach is that it can't be used for general purpose load and stores, it needs pre-computed parity. Storing parity is also an overhead in memory [6]. Parity check of value stored in **r4** is shown in Figure 2.20a.

- **Complementary Double check**: Conditions that rely on protected data are often targets of attacks. A double check that's complementary should be implemented in this case, it requires the attacker to perform 2 different attacks in a short time, making a successful fault injection hard.

```
 1. ldr r4,[r1];
 2. mov r12,r4;
 3. lsr r0,r4,#4;
 4. eor r12,r12,r0;
 5. lsr r0,r12,#2;
 6. eor r12,r12,r0;
 7. lsr r0,r12,#1;
 8. eor r12,r12,r0;
 9. and r12,r12,#1;
10. ldr r6,[r7];
11. cmp r12,r6;
12. bne <error>;
```

(a) Parity Checking

```
; the push{} instruction is equivalent
; to the stmdb sp!,{} instruction
stmdb sp, {r1, r2, r3, lr}
stmdb sp, {r1, r2, r3, lr}
sub   rx, sp,#16
sub   rx, sp,#16
mov   sp, rx
mov   sp, rx
```

(b) Instruction Replacement Sequence

Figure 2.20: Algorithmic instruction level techniques

- **Fault tolerant instruction sequences**: To specifically protect against instruction skip attacks, which converts an instruction equivalently to a **NOP**. An instruction is replaced by a sequence of instructions, that are still functional if an instruction is skipped. The paper [54] implemented this for ARM thumb 2 instructions. Replacement sequences for different instruction classes were proposed. Instruction replacement sequence for **push {r1,r2,r3,lr}** is shown in Figure 2.20b.

- **Iteration Check**: Loops execute a piece of code multiple times based on a condition. They can be attacked to terminate halfway without completing the required iterations either in order to bypass checks at the end of the loop or to gain access of intermediate data. Adding a loop check code that verifies the number of iterations run by the loop once it terminates handles this fault.

- **Random Code Delays**: Fault injections target particular instructions in the code and assume their position in time is always fixed in any simulations. Encapsulating a sensitive function or data by calls to random delays, goes against the above assumption and is effective against temporal fault attacks [76].

- **Execution flow counters**: A common fault attack is to corrupt execution flow by causing a branch to a attack function and returning back. Maintaining counters that help to track which functions are taken how many times can avert this attack. Maintaining and incrementing counters to identify functions uniquely causes significant performance overhead and is not scalable to large applications.

Modular exponentiation Algorithm

The Modexp operation, mathematically represented as $\mathbf{m^d}$ **mod N** is used in many cryptographic algorithms such as RSA, DSA and Diffie-Hellman [26]. Algorithmically, the performance of Modexp operation determines the cryptographic systems performance. The exponent is decomposed into a base 2^k number and Modexp operates on these bits from the most significant end (MSB). N-bit multiplications and reductions mod N form the critical operations in Modexp. Depending on the exponent bit, multiplications are either squarings, which are performed using faster squaring code or plain multiplications. The Modexp operations algorithm is as shown in Figure 2.21, the **if** condition can be injected with faults to corrupt the algorithm or timing difference can be used to perform a side channel attack.

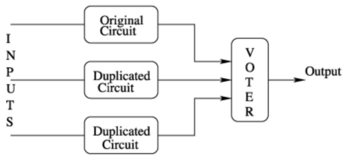
```
MODEXP($m, d, N$)
    INPUT: $m, d = (d_{\ell-1}d_{\ell-2}\cdots d_1d_0), N$
    OUTPUT: $m^d \bmod N$
1   $r \leftarrow 1$
2   for $j \leftarrow \ell, \ldots, 0$
3       $r \leftarrow r^2 \bmod N$
4       if $d_j \neq 0$                              // exponent bit set?
5           $r \leftarrow r \cdot m \bmod N$
6   return $r$
```

Figure 2.21: Modular exponentiation operation

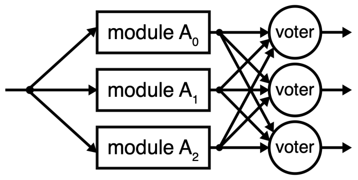
2.5.2. Hardware Countermeasures

Redundancy in hardware modules can be used for fault detection. Extra circuitry such as a voting logic or encoder and decoders in case of using error correcting codes can be used for fault correction. Choosing the suitable hardware countermeasure depends on the trade-off between allowable hardware overhead and level of fault/error correction required.

- **Triple Modular Redundancy**: This uses 2 extra ALUs and same instructions are computed on all the three ALU at the same time [49]. A majority voting logic is used to find the correct result. Fault in one processor is masked by results from other two. A configuration with single or triple voter for improved fault hardening can be used.



(2.22a) TMR with one voter



(2.22b) TMR with three voter

- **Residue Code**: For error detection of arithmetic operation, residue codes can be used. Since they don't work for Boolean operations and as error correction is required, and extra ALU performing the same ALU operation on the residues, instead of data is used [74]. The original result, modulo m and residue processor result are compared to catch a fault.

- **BCH Code**: A (63,36) BCH code can be used to encode and decode data from ALU and registers respectively to implement an ALU that can correct upto 5 faults injected at the same time [73]. Hardware overhead was experimentally shown to be **75%** of the original area, which is considerably less compared to TMR, which causes **200%** area overhead.

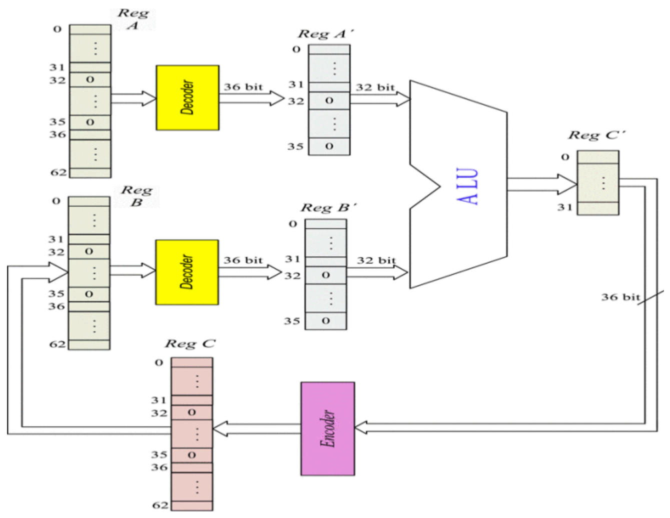


Figure 2.23: Fault tolerant ALU using BCH Code

- **Watchdog**: A watchdog processor can be used to ensure fault free data transmission between memory and the main processor [9]. The watchdog sits on the external memory bus and listens to all the transactions on the bus. Watchdog saves a shadow copy of all the writes to memory and on a read, compares its value with value returned by memory. On a mismatch a transaction is made to repeat. Control flow check is also discussed in the literature, which uses signature schemes to monitor code blocks. This can't be scaled to large real world applications. A new approach is proposed in Section 5.5.2.

3
RISC-V Processors

3.1. RISC-V

RISC-V is a reduced instruction set architecture based open ISA, which supports 32,64,128 bit data widths [13]. RISC-V is increasingly being adopted in industry and academia since it's introduction in 2010. The following factors make it more attractive compared to other ISAs:

- As it's open source and incurs no licensing costs when compared to other similar ISAs, widespread chip design and usage by various stakeholders is possible.
- Base ISA, which specifies concepts such as instruction encoding, address modes, integer arithmetic is frozen, allowing software developers to design tools such as compilers.
- Standard extensions which provide additional functionality along side with base ISAs, can be added and standardised. The new added extensions are designed to work with other existing extensions.
- High configurability of ISA allows it to be suited for various range of applications from low power to high performance with support for dedicated accelerators.

All memory accesses are byte level addressable and data is stored in little endian format in the memory. All the instructions must be aligned at 32-bit boundaries. Conditional codes for instructions other than branches are not supported as well as carry out bits to detect overflows. Three levels of privileged modes are defined, **machine, hypervisor and supervisor**, each with their own control & status registers.

3.1.1. Instruction Set Architecture

RISC-V has a base ISA that should be present in any implementation and optional extensions can be added. The base ISA is designed to have minimum set instructions with an optional support to variable length instructions. An ISA is characterised by the width of the integer register. If the instructions are 32 bit, they should be aligned to 32 bit boundaries.

The base ISA has 32 registers and each of them have a standard function defined for them according to Application Binary Interface (ABI). ABI allows programs to access system hardware in order to ensure software interoperability. Development tool-chains refer ABI names for convenience as opposed to hard-coded register numbers. The instructions in RISC-V RV32I base ISA are shown in Appendix B.

An unusual condition at runtime due to an executing instruction is termed as an exception and is synchronously handled. The handler executes in a privileged environment. An interrupt on the other hand, is due to an external event and occurs asynchronously to the executing instruction. Four core instructional formats are possible, namely R/I/S/U, shown in Figure 3.1.

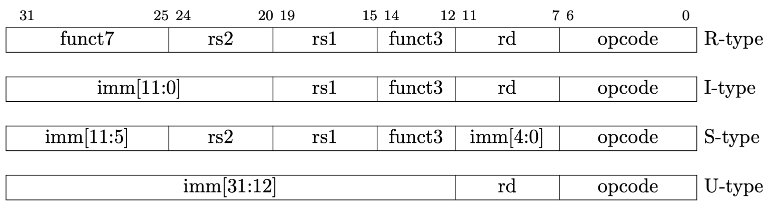
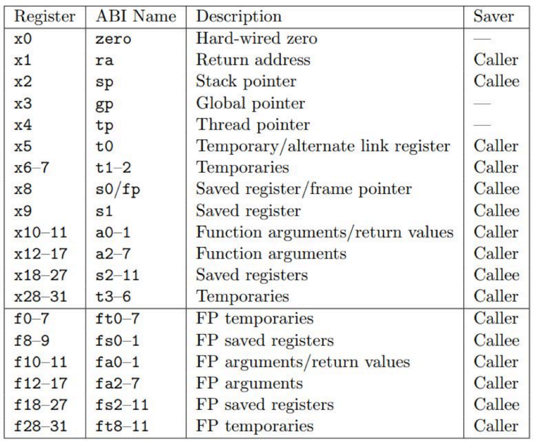


Figure 3.1: RISC-V Base Instruction Formats

The base ISA is integer instruction set with 32-bit(RV32I) , 64-bit(RV64I), 128-bit(RV128I) data width specifications defined. Multiple extensions to base ISAs are possible and some frequently used ones are described below [42],

- M - Addition of a Multiply or Divide unit.
- F - To include a Single point precision floating point unit.
- D - To add a Double precision floating point unit.
- A - Support for atomic operations.
- C - Compressed ISA.

RISC-V defines **hart** as a hardware thread that contains its full set of architectural registers and is managed completely by hardware. It essentially is an abstraction of a core to support multi-threading. Any execution environment can have one or more harts and they are completely transparent to the environment. Harts execute independently, so they fetch and execute instructions independent of other harts. In a software execution environment, from a perspective of a user program, a 'RISC-V system', constitutes of a hart and its associated memory [3].



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figure 3.2: RISC-V ABI

3.1.2. Control and Status registers

CSRs store information related to various units such as counters, timers and floating point. Most of them are used by privileged code. A total of 4096 control and status registers can be defined per hart. Some CSRs are unique per hart, but some are shared. They can be read and written to only by special instructions, even though they are memory mapped. Access to CSRs is also dependent on the privilege level of the software, some have restricted access at lower privileged levels, other such as mscratch and mepc are duplicated. Functionality of only some CSRs is specified and others are implementation defined to be used by designers for additional functionality. Access to CSRs in a hart are performed in program order. The following specify some atomic instructions used to access CSRs,

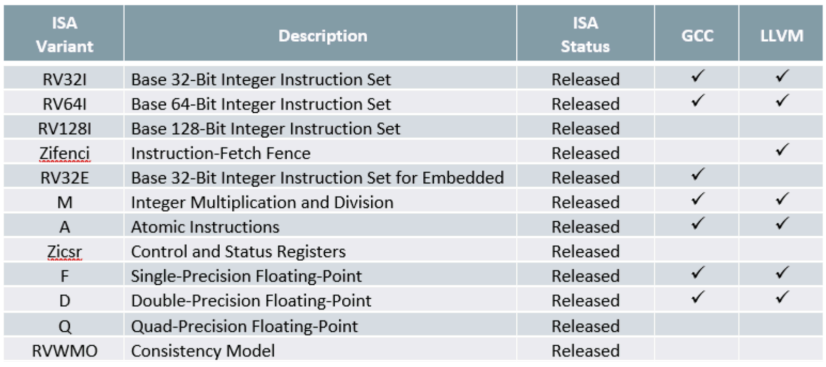
- **CSRRW** - Read zero extended CSR value to $\mathbf{r_d}$, write it with value in $\mathbf{r_s}$.
- **CSRRC** - Read zero extended CSR value to $\mathbf{r_d}$, bits set in $\mathbf{r_s}$ are treated as bit mask and corresponding bits in CSR are cleared.
- **CSRRSI** - zero extended 5-bit unsigned immediate is written to CSR.

Some frequently used CSR registers are as below,

- **mhartid** - Specifies the hardware thread ID in integer format.
- **mcycle** - Counter for machine cycles.
- **tdata1** - Trace data register.
- **dpc** - Debug PC.
- **mie** - Machine interrupt enable register.
- **fflags** - Floating Point exceptions.
- **hpmcounter** - Performance monitoring counter.

3.1.3. RISC-V Tool Chain

A tool chain contains of compiler, assembler, linker, debugger and libraries required to convert an application in high level language to a binary, that can enable it to run on the actual microprocessor[67]. RISC-V tool chains are currently provided by two open source frameworks, **GCC** and **LLVM**. Code can be compiled to all RISC-V base ISAs and its extensions by GCC. Systems with RTOS such as linux and systems with no RTOS (Bare metal) can both use GCC. LLVM provides compiler back-end and libraries for linux, but bare-metal support is lacking. **GDB** can be used for debugging and **Clang**, which uses LLVM as its back-end can also be used as a compiler. LLVM though offers other advantages such as support for new programming languages like **Rust**. LLVM can also be used for design exploration as it addition of custom instructions and compiler optimisations are allowed .**Yocto** an embedded linux distribution was ported for RISC-V[46]. The **OpenSBI** project provides a supervisor binary interface and standardises linux system development. Support for various ISAs by GCC and LLVM are shown in the table.



ISA Variant	Description	ISA Status	GCC	LLVM
RV32I	Base 32-Bit Integer Instruction Set	Released	✓	✓
RV64I	Base 64-Bit Integer Instruction Set	Released	✓	✓
RV128I	Base 128-Bit Integer Instruction Set	Released		
Zifenci	Instruction-Fetch Fence	Released		✓
RV32E	Base 32-Bit Integer Instruction Set for Embedded	Released	✓	
M	Integer Multiplication and Division	Released	✓	✓
A	Atomic Instructions	Released	✓	✓
Zicsr	Control and Status Registers	Released		
F	Single-Precision Floating-Point	Released	✓	✓
D	Double-Precision Floating-Point	Released	✓	✓
Q	Quad-Precision Floating-Point	Released		
RVWMO	Consistency Model	Released		

Figure 3.3: LLVM and GCC supported ISAs

RISC-V specific command line arguments for GCC start with **-m**. The frequently used ones are as follows,

- **-march** - To select the ISA for which, the assembler generates the assembly instruction for. Instructions and registers for the compiler to use are also specified by it.
- **-mabi** - To choose a specific **ABI**, that dictates the calling conventions.
- **-mtube** - Added to alter the performance of the code by including optimisations particular to a specified micro-architecture.

3.2. RISC-V OpenSource Cores

3.2.1. PicoRV32

A 32-bit processor with a small footprint of 750 - 2000 LUTs, which is designed especially to be a auxiliary core for FPGA designs. It is a size optimised core designed for simple embedded applications. Higher frequencies up-to 450MHz are supported by the processor and thus can be added to designs without crossing clock domains [19]. The processor implements RV32I, RV32E ISA and is highly configurable. Various configurations such as enabling dual port register file, barrel shifter, compressed ISA, fast multiplier & Divider and IRQ support are possible. An optional co-processor interface is also supported. The previous configurations are selected based on intent of design, be it either low power, low area or high performance. To communicate either with the external peripherals or local RAM, ROM including other processors in a multi processor system, low latency BRAM interface and AXI4 are supported [22].

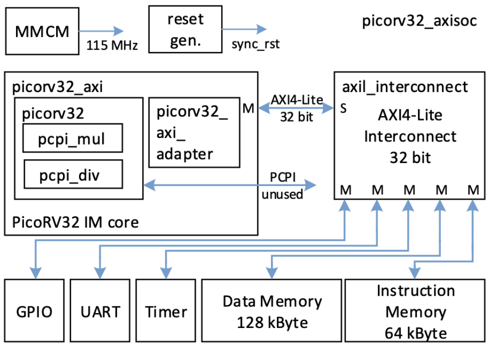
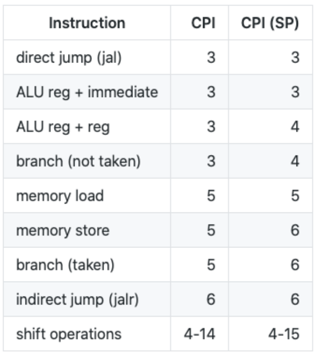


Figure 3.4: PicoRV32 IP Core Block Diagram [13]

PicoRV32 is a simple single pipeline stage *von Neumann* processor, designed in Verilog HDL. So, to access the instruction and data memory a single interface is used. Multiplier and divider use co-processor interface (PCPI) to connect to the base processor. It has a CPI (Clocks Per Instruction) of 3 to 6 clock cycles for simple integer arithmetic, memory operations and branches [59]. Shifts take 15 clock cyles without a barrel shifter and 4 cycle with it. Multiply, divide and reminder take 40 cycles, whereas **MULH**, both signed and unsigned take 72 cycles. PicoRV32 does not support out of order execution nor does it have a memory management unit and a FPU. It doesn't have JTAG or any other debug peripheral support. Support for either instruction cache or data cache is also absent [63]. An active low synchronous reset for flip-flops and memories is used. A single clock domain with rising clock edge is used to synchronise. For SoC designs based on PicoRV32, that use local memory and several peripherals, it requires an intermediate interconnect as PicoRV32 supports only single AXI interface [13]. Support for both scratch memory and system bus configurations is provided, though tightly coupled scratch memory support is lacking [25]. A CPI of 4 is achieved on Dhrystone benchmark with look-ahead interface and 5.23 without it.



Instruction	CPI	CPI (SP)
direct jump (jal)	3	3
ALU reg + immediate	3	3
ALU reg + reg	3	4
branch (not taken)	3	4
memory load	5	5
memory store	5	6
branch (taken)	5	6
indirect jump (jalr)	6	6
shift operations	4-14	4-15

Figure 3.5: PicoRV32 CPI

3.2.2. DarkRISCV

DarkRISCV is a RV32I RISC-V implementation targeting Spartan-6 FPGAs[1]. A CPI of almost '1' is achieved for almost all instructions, except for missed branches, which causes a bubble for one clock cycle due to pipeline flush. It is a compact design in obfuscated Verilog, available on a BSD license. A SoC with DarkRISCV with a support for cache controllers and glue logic is also available. Some design considerations are,

- An option to select between a 2-stage and 3-stage pipeline versions. The 2-stage pipeline has an IPC (Instructions Per Clock) of 0.85 and can run at 50MHz. The 3-stage pipeline has an IPC of 0.7 and can run at 100MHz, thus has a higher performance than the former.
- In the 2-stage pipeline, the first clock is spent in fetch and the second is decode and execute. The pipeline is overlapped without interlocks to give a high IPC.
- BlockRAMs need two clock cycles to return the data. In the first clock, the address is latched and in the second, the Data. Thus, for the 2-stage pipeline to work it requires a faster memory, typically implemented in LUTs. To use BlockRAM based Caches or external memories, the 3-stage pipeline was designed. Pipeline stages do a Fetch, Decode and Execute correspondingly.
- To integrate a cache controller easily, Harvard architecture was chosen.
- A pipeline version which operates on both clock edges was also designed, but is not used as it requires a ROM or RAM to operate on dual phase clock.
- Branch predictors or delayed branches are not implemented.
- **FENCE** and **CSR*** instructions are not implemented.
- Coarse grained-multithreading and 16x16 bit MAC instructions are optionally available.
- A compact design that takes only around 1000 **LUTs** on Spartan FPGAs.
- Works with **GCC** and has an optional **RV32E** support.

Features such as GPIO, Ethernet controller, NOC and others are under development and Dark-RISCV road-map is shown in Figure 3.6.

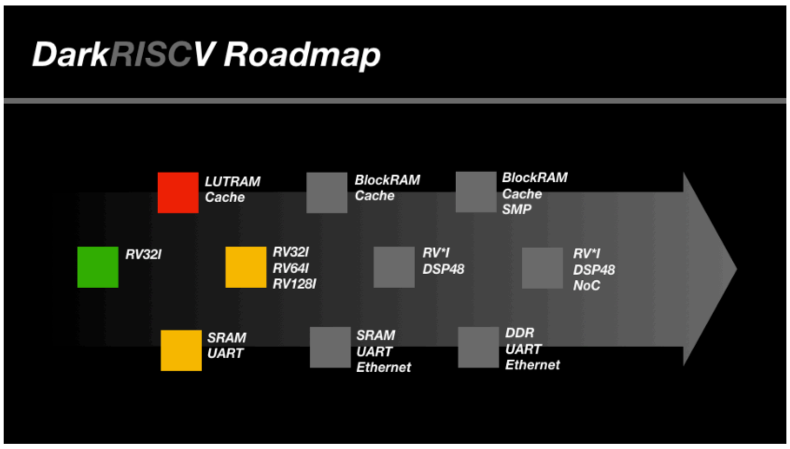


Figure 3.6: DarkRISCV Roadmap

4 Methodology

4.1. Methodology Overview

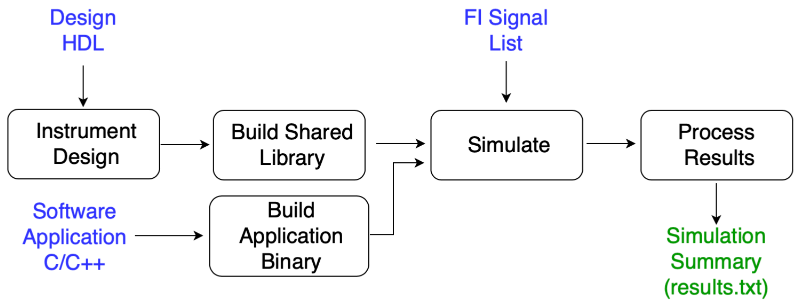


Figure 4.1: Top level Methodology

The methodology to instrument the design and simulate fault can be broadly broken into the tasks shown in Figure 4.1. A design based on Verilog HDL is taken as an input and summary of simulation results is generated. A brief description of each step is shown below and a detailed description of the same is given in the following sections.

- **Instrument Design**: The HDL design is pre-processed to select the required configuration and remove constructs that can't be handled by the parser. Next, the code parser converts the design to an AST(Abstract syntax Tree) and a custom parser adds signals and ports to the tree to enable fault injection. Finally, the tree is converted back to HDL design by code generator.
- **Build Shared Library and Application Binary**: The binary of the application to be run on the processor is generated by RISC-V toolchain. The instrumented design obtained from the previous step is converted to a shared library to facilitate being accessed from python simulation framework.
- **Simulate**: A simulation framework uses the previously generated simulator shared library and application binary along with a list of signals to be injected with fault that is inputted from the user, to simulate the design and generate processor trace.
- **Process Results**: As manual inspection of the generated processor trace log is error prone, a summary of the fault injection campaign is generated. The summary includes the total passing/ failing simulations, list of error signatures encountered along with the simulation log name. Simulation that don't fall into any of the fault class are denoted with the mismatch they fail with.

4.2. Tool Flow Overview

The framework reads Verilog based design and other configuration input files, generating the run summary listing the passing and failing tests along with error type and test name. Dhrystone benchmark is the application that is run on the processor while injecting faults. Dhrystone was chosen as it serves as a good representation of actual integer workloads run by processors. It has all the different classes of instructions in the ISA, all of which can potentially be targeted by an attacker. Various tools such as yosys, pyverilog, and verilator are integrated into the framework. RISC-V GNU toolchain was used to compile applications to binaries. Both design instrumentation framework and simulation framework are written in python. The instrumented design is converted to a C++ model to avoid using commercial simulators and to obtain large speedup in simulations achievable by verilator. The tasks achieved by the complete tool flow can be can broken down into fault instrumentation of the design and simulation of the design with fault injection. The tool flow overview is shown in the Figure 4.2.

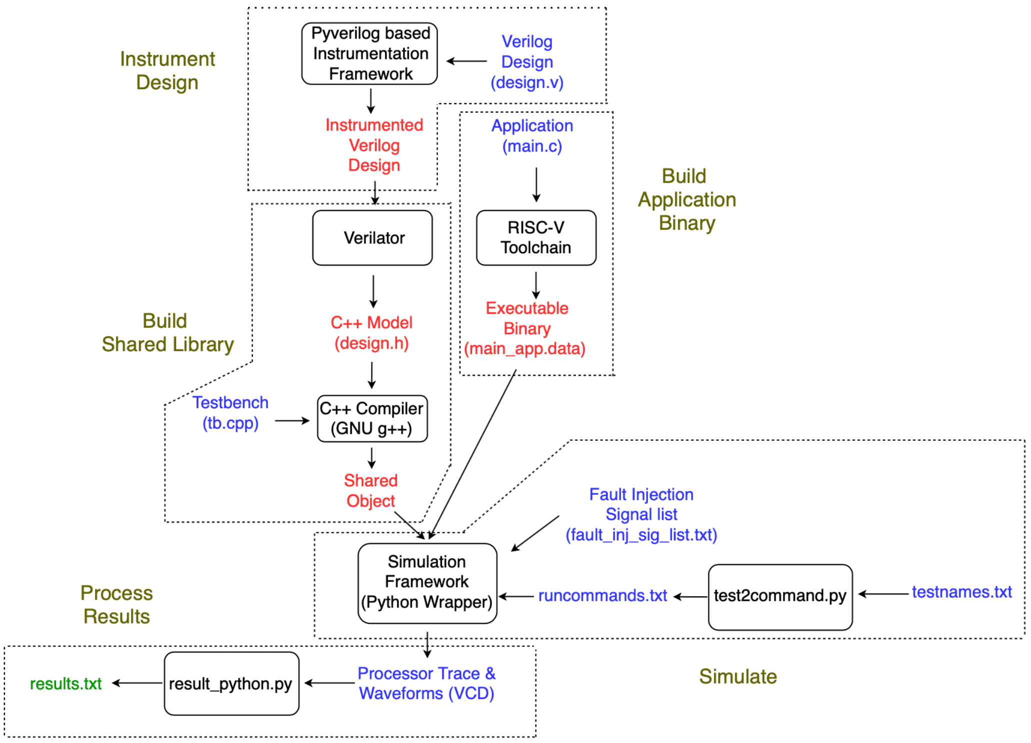


Figure 4.2: Flow Overview

Verilog design files are read by the design instrumentation framework which preprocesses it and adds a port to the top level module of the design which has fault injection control lines. This instrumented design is converted to a C++ model by verilator and in turn into a shared object by **g++** compiler. The simulation framework uses this simulator object to inject faults in run-time and generates a report of the run finally. If the names of the tests to be run are already known from a previous fault campaign run, **test2command.py** can be used to generate corresponding commands to be directly used in **python_wrapper.py**. The signals into which faults are to be injected are specified by a fault injection signal list file. The simulation framework injects a single fault per simulation for all the signals specified in the fault injection signal list file. Both processes are described in-depth in the following sections.

4.3. Instrument Design

Pyverilog, a hardware design processing toolkit for Verilog, is used for design instrumentation. It provides code parser, data/control flow analyser and code generator to create custom code analysers, translators and generators for Verilog [71]. To instrument the design, mutant technique is used, where the original design module is replaced by a mutant module. The mutant module behaves like the original one when inactivate and can be activated by driving a corresponding fault control lines[37]. The top level overview of the flow is shown in the Figure 4.3.

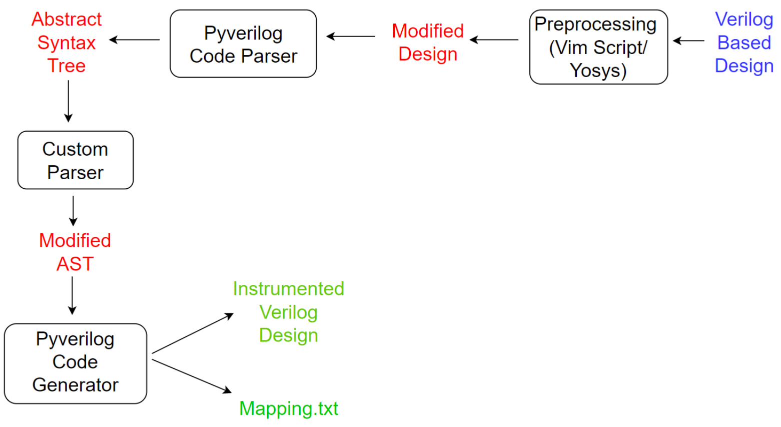


Figure 4.3: Design Overview

Pyverilog code parser cannot handle the following constructs in Verilog,

- **‘assert, ‘ifdef, generate** statements.
- Spacing between numericals such as **3’b 001**.
- Synthesis directives such as **full_case** and **parallel_case**.
- Commented modules and their port declarations.

Since none of them effect the design intent of the module, a preprocessing step, to remove them is necessary. A **Vim script** is written to perform preprocessing on the design. **Yosys** along with configuration file (config.vh) is used to configure the design according Verilog parameters as **‘ifdef** can’t be handled by pyverilog. The resulting modified design is passed to the Pyverilog code parser that converts it into an abstract syntax tree (AST). The AST is hierarchical structure with each node representing a program construct in the source code. Nodes could be **module definitions**, **port lists**, **if statements**, **always blocks** etc. Nodes can have children, for example, **non-blocking assignment** can have an Lvalue and Rvalue. **Lvalue** and **Rvalue** can further have children like identifiers. Nodes in Pyverilog are implemented as classes and once it encounters a similar program construct in the source code, it creates an object of that class. AST essentially is an hierarchical structure of objects. The base object of an AST is **Source**, which instantiates **Description**, which in turn has children of different module definitions (**ModuleDef**). All the classes in Pyverilog have **children** method, which return all its children as a list or tuple. The **__init__** method in all classes, as a normal python coding practice, initialises properties of the classes, here mostly their child classes. As an example, Verilog code for a D flip flop is show in Figure 4.4a. When Pyverilog parser reads the design, it generates an AST as shown in Figure 4.4b.

```
01 module dff
02 ( input clk,
03  input rst_n,
04  input Dvalue,
05  output reg Qvalue);
06
07 always @(posedge clk) begin
08
09 if(~rst_n)
10  Qvalue <= 1'b0;
11 else
12  Qvalue <= Dvalue;
13
14 end
15
16 endmodule
```

(a) D-Flip Flop Verilog Code

```
Source:  (at 1)
 Description:  (at 1)
  ModuleDef: dff (at 1)
   Paramlist:  (at 0)
   Portlist:  (at 2)
    Ioport:  (at 2)
     Input: clk, False (at 2)
    Ioport:  (at 3)
     Input: rst_n, False (at 3)
    Ioport:  (at 4)
     Input: Dvalue, False (at 4)
    Ioport:  (at 5)
     Output: Qvalue, False (at 5)
     Reg: Qvalue, False (at 5)
   Always:  (at 7)
    SensList:  (at 7)
     Sens: posedge (at 7)
      Identifier: clk (at 7)
    Block: None (at 7)
     IfStatement:  (at 9)
      Unot:  (at 9)
       Identifier: rst_n (at 9)
      NonblockingSubstitution:  (at 10)
       Lvalue:  (at 10)
        Identifier: Qvalue (at 10)
       Rvalue:  (at 10)
        IntConst: 1'b0 (at 10)
      NonblockingSubstitution:  (at 12)
       Lvalue:  (at 12)
        Identifier: Qvalue (at 12)
       Rvalue:  (at 12)
        Identifier: Dvalue (at 12)
```

(b) AST of D-Flip Flop

Figure 4.4: D-Flip Flop Verilog code and AST

Custom Parser

Custom parser automatically modifies the following syntactical units in the behavioural description,

- Wires, Registers, Register Files, Memories.
- Conditions such as ternary operators and if-else.
- Continuous assignments (assign) and procedural assignments.
- Sensitivity lists.
- Case statement control expression and individual cases.

The parser was designed in python and supports **Verilog-2001** completely apart from some specific syntactical constructs like **generate statements**, along with partial support for **Verilog-1995**. The port declaration of **Verilog-1995** is not supported, only **Verilog-2001 ANSI-C** style works. Signals using system calls like **$signed** can't be injected with a fault using this parser. Fault instrumentation of hierarchical Verilog modules, that are in separate files is buggy and requires some manual effort to validate the instrumented design. The top level algorithm of the custom parser is described in Algorithm 1. The parser performs a **depth first traversal** of the AST to recursively add the fault injection control lines. If a module name to be instrumented is not specified, the top level module in the design is initially identified and hierarchical tree of the modules is built. All the modules are then instrumented from leaf modules to top most module, with their injection busses moving up the hierarchy. If **parameters** are used in to define widths of wires, registers or dimensions in memories and register files, they will be replaced by the pre-processing script to an integer number before it is passed to the parser, as it can't be handled. Tools like Pycharm and Jupyter Notebooks were used as IDEs to design the parser.

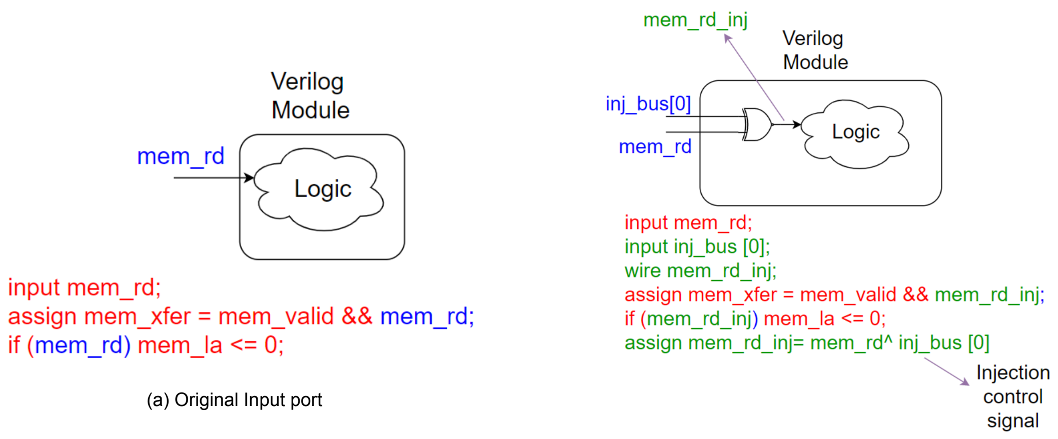
Algorithm 1 Custom Parser Algorithm

```
for ast.description.children() do
                                                // Iterate over all children of description
    if ModuleDef then
        for ModuleDef.children() do
            if Portlist then
                for Portlist.children() do
                    if IoPort then
                        if len(IoPort.children()) == 2 then
                            Make the Output a Wire      // If 2 children, Its an output of type reg
                            Delete the Reg Object
                        end
                        for IoPort.children() do
                            if Object is an Input or Output then
                                Extract signal width and name      // Object is an Input or Output port
                                Create an Injection Wire Object                                    //
                                Replace name with injection name recursively in all the design    //
                                Add Injection Wire Object declaration to Design                    //
                                Update counter of Injection bus                                    //
                            end
                            if Object is Output Reg then
                                Extract signal width and name   // Object is a output of register type
                                Create an Injection Object (_internal)                             //
                                Replace name with injection name recursively in all the design    //
                                Add Reg declaration and assign statement                           //
                                Update counter of wires, Injection bus                             //
                            end
                        end
                    end
                end
            end
            if Declaration then
                for Decl.children() do
                    if Object is a wire then
                        Extract signal width and name                          // Object is a wire
                        Create an Injection Wire Object                                    //
                        Replace name with injection name recursively in all the design    //
                        Add Injection Wire Object to Injection bus list                    //
                        Update counter of Injection bus, wires                             //
                    end
                    if Object is a Reg then
                        if Reg corresponds to an Output then
                            break
                        else
                            Extract width of Reg                           // Object is a Register
                            Create an Injection Bus Object                                     //
                            Add Injection bus object in design where Reg is found              //
                            Update counter of Injection bus, wires                             //
                        end
                    else
                        Extract dimensions, width              // Object is a RegFile or Memory
                        Special Cases for different declaration styles                     //
                        Create an Injection bus Object for dimensions, width               //
                        Add Injection bus Object recursively in design                     //
                        Update counter of Injection bus                                    //
                    end
                end
            end
        end
    end
end
Parse AST again, add inj_bus to portlist of the module
```

The fault model considered is bit flip, so a **XOR** gate is used to bit flip the value of signals or registers. Further, fault instrumentation of various syntactical structures in Verilog is discussed.

Input Port

An input to a Verilog module is always of type wire and can be defined to of any width. Pyverilog has a **Input** class defined to handle input types. An input can be used in the design in assignments such as continous and procedural as well as sensitivity lists, conditions for case, if-else etc. Input is never driven in the design, so it is never on the **LHS** of a statement in the design. For the instrumentation of the design, parser creates a wire, with name **_inj** prefix of the actual input name and an injection control bus to control fault injection. Injection control bus and input wires are **XORed** and the resultant signal is used everywhere in the design to replace the actual input recursively. The **assign** statement which performs this is added to the design. An example is shown in the Figure 4.5.



```
input mem_rd;
assign mem_xfer = mem_valid && mem_rd;
if (mem_rd) mem_la <= 0;
```

(a) Original Input port

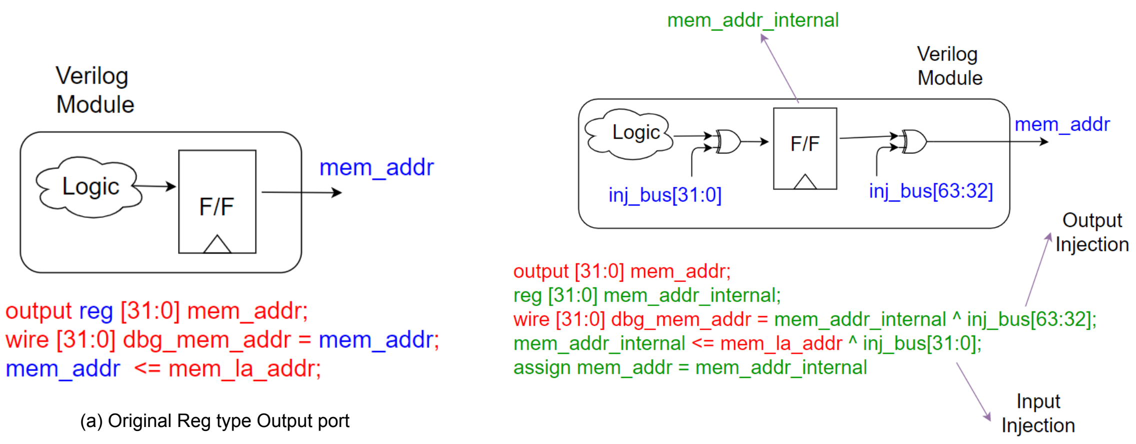
```
input mem_rd;
input inj_bus [0];
wire mem_rd_inj;
assign mem_xfer = mem_valid && mem_rd_inj;
if (mem_rd_inj) mem_la <= 0;
assign mem_rd_inj= mem_rd^ inj_bus [0]
```

(b) Instrumented Input port

Figure 4.5: Input port instrumentation

Output Port of type Reg

In this case, Pyverilog has a class Output. The output is converted to a wire from reg type and a reg with same dimension of actual output with name **_internal** prefixed is created and used everywhere in the design instead of the actual output. Injection buses at the input and output of this reg are created, which are added as **XORs** at **LHS** and **RHS** respectively.



```
output reg [31:0] mem_addr;
wire [31:0] dbg_mem_addr = mem_addr;
mem_addr  <= mem_la_addr;
```

(a) Original Reg type Output port

```
output [31:0] mem_addr;
reg [31:0] mem_addr_internal;
wire [31:0] dbg_mem_addr = mem_addr_internal ^ inj_bus[63:32];
mem_addr_internal <= mem_la_addr ^ inj_bus[31:0];
assign mem_addr = mem_addr_internal
```

(b) Instrumented Reg type Output port

Figure 4.6: Reg type Output port instrumentation

Output Port of type Wire

A wire with **_inj** prefixed to actual output name is created and replaces all instances of actual output in the design. This wire Xored with injection control bus is assigned to actual output.

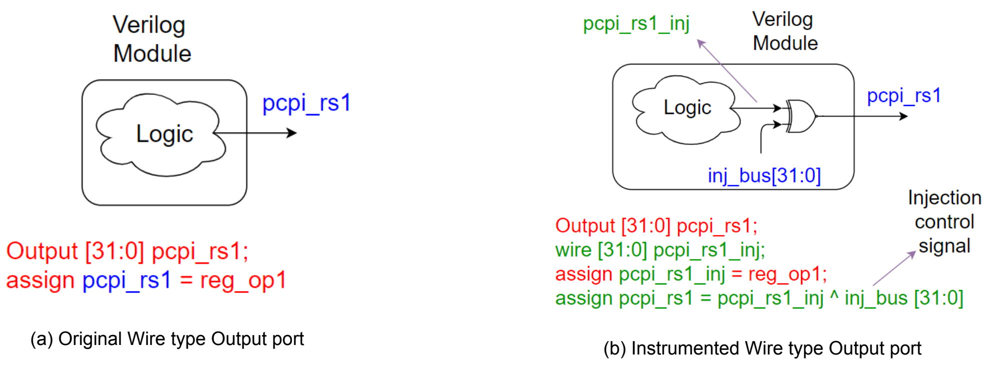

Verilog Module
Logic
pcpi_rs1
Output [31:0] pcpi_rs1;
assign pcpi_rs1 = reg_op1
(a) Original Wire type Output port
pcpi_rs1_inj
Verilog Module
Logic
pcpi_rs1
inj_bus[31:0]
Injection control signal
Output [31:0] pcpi_rs1;
wire [31:0] pcpi_rs1_inj;
assign pcpi_rs1_inj = reg_op1;
assign pcpi_rs1 = pcpi_rs1_inj ^ inj_bus [31:0]
(b) Instrumented Wire type Output port

Figure 4.7: Wire type Output port instrumentation

Reg Types

Similar to the output port of type reg, busses to inject fault both at input and output are added on **LHS** and **RHS** of a statement where reg type is being driven or driving a variable respectively. Only a slice of the reg can be assigned to, in some cases and the parser is designed to handle such scenarios.

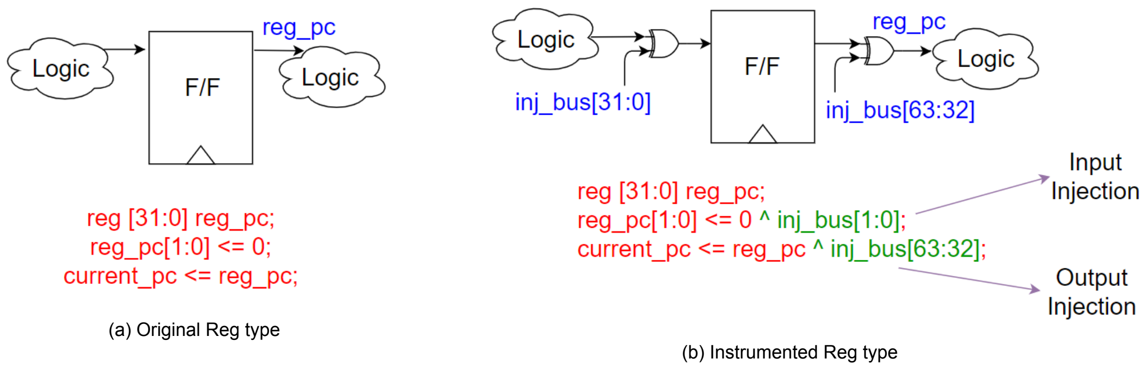
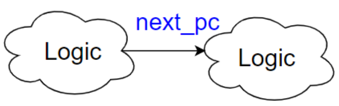


Figure 4.8: Reg type instrumentation

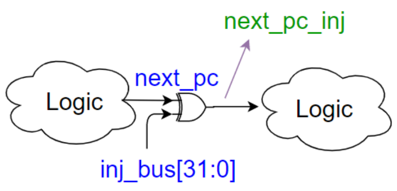
Wire Types

Like the output port of wire type, a wire with **_inj** prefix to actual wire is created. This wire is used to replace recursively all the occurrences of the actual wire, if its only on the **RHS** of a statement as shown in Figure 4.9. The instances where its on the **LHS** are not modified. A special case for a wire type is when it used to connect to a module that is instantiated in an another module. If the wire is used to connect to an output of a module, its instance is not altered, where as an input instance is replaced by the **_inj** version as shown in Figure 4.10.



```
wire [31:0] next_pc;
wire la_first = prefetch && next_pc[1];
assign la = do ? {next_pc[31:2], 2'b00} : 0;
dbg_ins_addr <= next_pc;
assign next_pc = reg_out;
```

(a) Original Wire type



```
wire [31:0] next_pc;
wire [31:0] next_pc_inj;
wire la_first = prefetch && next_pc_inj[1];
assign la = do ? {next_pc_inj[31:2],2'b00}:0;
dbg_ins_addr <= next_pc_inj;
assign next_pc = reg_out;
assign next_pc_inj = next_pc ^ inj_bus[31:0];
```

Injection
control
signal

(b) Instrumented Wire type

Figure 4.9: Wire type instrumentation

Module
Output

```
wire tst1;
Adder A1 (...., tst1);
assign tst4 = tst1 ^ tst5;
```

Injection
control
signal

```
wire tst1;
wire tst1_inj;
Adder A1 (...., tst1);
assign tst4 = tst1_inj ^ tst5;
assign tst1_inj = tst1 ^ inj_bus[0];
```

(a) Instrumentation of a wire, that's an output from a module

Module
Input

```
wire tst1;
Adder A1 (tst1,....);
assign tst1 = tst4 ^ tst3;
```

Injection
control
signal

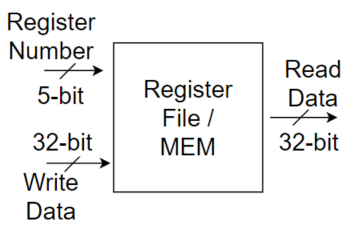
```
wire tst1;
wire tst1_inj;
Adder A1 (tst1_inj,....);
assign tst1 = tst4 ^ tst3;
assign tst1_inj = tst1 ^ inj_bus[0];
```

(b) Instrumentation of a wire, that's an input to a module

Figure 4.10: Instrumentation of wires to and from a module

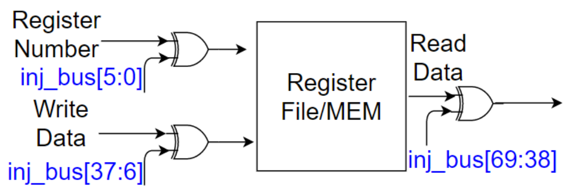
Register Files and Memories

Register Files and Memories have both dimensions and width. Similar to the case of a register, fault injection lines are added both on and instances of a register/memory in a statement for write and read respectively. An injection bus for address is also created and is used both on **LHS** and **RHS** instances of the register/memory. No other extra wires or registers are created.



```
wire [31:0] cpuregs[0:31];
cpuregs[latched_rd] <= cpuregs_wrdata;
cpuregs_rdata <= cpuregs[decoded_rs1];
```

(a) Original Register File / Memory



Address
Injection

Output
Injection

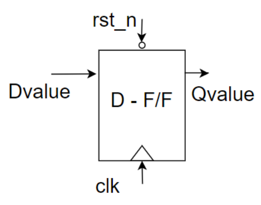
```
wire [31:0] cpuregs[0:31];
cpuregs[latched_rd ^ inj_bus[5:0]] <= cpuregs_wrdata ^ inj_bus[37:6];
cpuregs_rdata <= cpuregs[decoded_rs1^inj_bus[69:38]];
```

Input
Injection

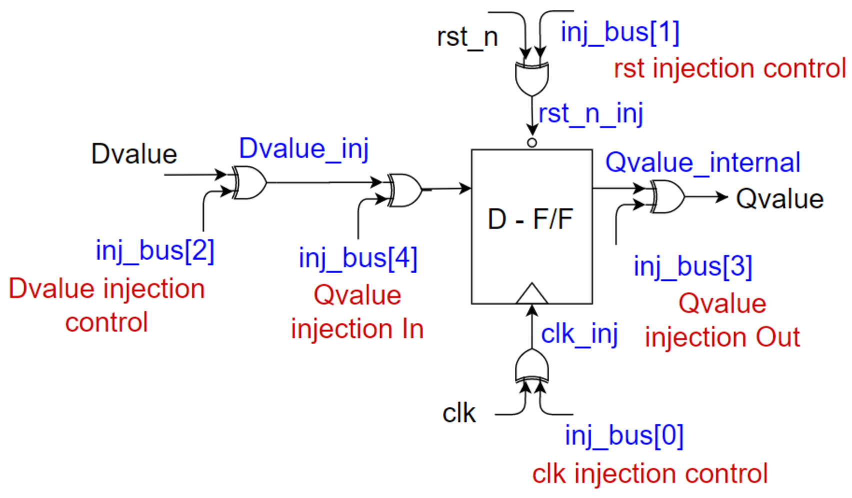
(b) Instrumented Register File / Memory

Figure 4.11: Register File / Memory instrumentation

A D-F/F shown in Figure 4.12a whose Verilog code is mentioned in Figure 4.4a. When it is processed by the custom parser, the resulting design with **XOR** gates and fault injection control line shown in Figure 4.12b is obtained.



(a) D-Flip Flop



(b) Instrumented D-Flip Flop

Figure 4.12: Fault instrumentation of a D-Flip Flop

The AST of the D-F/F is modified to include the fault control signals to instrument the design. It also produces a **mapping.txt** file that includes the names of the injection control signal that are added, the module to which they are added, the line number in the module where they were added and the index of the injection bus which they correspond to. This is shown in Figure 4.13.

```
Line no: Module Name : Original Signal Name: Injection Control Name ->Injection bus index
2: dff:clk: clk_inj_cntrl -> inj_bus[0]
3: dff:rst_n: rst_n_inj_cntrl -> inj_bus[1]
4: dff:Dvalue: Dvalue_inj_cntrl -> inj_bus[2]
5: dff:Qvalue: Qvalue_inj_out -> inj_bus[3]
5: dff:Qvalue: Qvalue_inj_in -> inj_bus[4]
```

Figure 4.13: Mapping file for D-Flip Flop

Code Generator

Code generator converts the modified AST obtained from the custom parser shown in Figure 4.14a back to Verilog code, shown in Figure 4.14b. The statements in red in AST are from original AST and the statements in green are added by the custom parser. It is similar in the Verilog code of the D-F/F shown. The obtained Verilog design has a fault injection bus port in it's top level module's port list that can be used to inject faults into the design. The design instrumentation framework can be used for behavioural level designs as well as gate level designs.

```
Source:  (at 1)
 Description:  (at 1)
  ModuleDef: dff (at 1)
   Paramlist:  (at 0)
   Portlist:  (at 2)
    Ioport:  (at 2)
     Input: clk, False (at 2)
    Ioport:  (at 3)
     Input: rst_n, False (at 3)
    Ioport:  (at 4)
     Input: Dvalue, False (at 4)
    Ioport:  (at 0)
     Input: inj_bus, None (at 5)
      Width:  (at 5)
       Constant: 4 (at 5)
       Constant: 0 (at 5)
    Ioport:  (at 5)
     Output: Qvalue, False (at 5)
   Decl:  (at 0)
    Wire: clk_inj, False (at 7)
   Decl:  (at 0)
    Wire: rst_n_inj, False (at 7)
   Decl:  (at 0)
    Wire: Dvalue_inj, False (at 7)
   Decl:  (at 0)
    Reg: Qvalue_internal, False (at 7)
   Always:  (at 7)
    SensList:  (at 7)
     Sens: posedge (at 7)
      Identifier: clk_inj (at 7)
    Block: None (at 7)
     IfStatement:  (at 9)
      Unot:  (at 9)
       Identifier: rst_n_inj (at 9)
      NonblockingSubstitution:  (at 10)
       Lvalue:  (at 10)
        Identifier: Qvalue_internal (at 10)
       Rvalue:  (at 10)
        Xor:  (at 10)
         IntConst: 1'b0 (at 10)
         Pointer:  (at 5)
          Identifier: inj_bus (at 5)
          IntConst: 4 (at 0)
      NonblockingSubstitution:  (at 12)
       Lvalue:  (at 12)
        Identifier: Qvalue_internal (at 12)
       Rvalue:  (at 12)
        Xor:  (at 12)
         Identifier: Dvalue_inj (at 12)
         Pointer:  (at 5)
          Identifier: inj_bus (at 5)
          IntConst: 4 (at 0)
   Assign:  (at 8)
    Lvalue:  (at 8)
     Identifier: clk_inj (at 8)
    Rvalue:  (at 8)
     Xor:  (at 8)
      Identifier: clk (at 8)
      Pointer:  (at 2)
       Identifier: inj_bus (at 2)
       IntConst: 0 (at 0)
   Assign:  (at 8)
    Lvalue:  (at 8)
     Identifier: rst_n_inj (at 8)
    Rvalue:  (at 8)
     Xor:  (at 8)
      Identifier: rst_n (at 8)
      Pointer:  (at 3)
       Identifier: inj_bus (at 3)
       IntConst: 1 (at 0)
   Assign:  (at 8)
    Lvalue:  (at 8)
     Identifier: Dvalue_inj (at 8)
    Rvalue:  (at 8)
     Xor:  (at 8)
      Identifier: Dvalue (at 8)
      Pointer:  (at 4)
       Identifier: inj_bus (at 4)
       IntConst: 2 (at 0)
   Assign:  (at 8)
    Lvalue:  (at 8)
     Identifier: Qvalue (at 8)
    Rvalue:  (at 8)
     Identifier: Qvalue_internal (at 8)
```

(a) Modified AST of D-Flip Flop

```
module dff
(
input clk,
input rst_n,
input Dvalue,
input [4:0] inj_bus,
output Qvalue
);

wire clk_inj;
wire rst_n_inj;
wire Dvalue_inj;
reg Qvalue_internal;

always @(posedge clk_inj) begin
if(~rst_n_inj) Qvalue_internal <= 1'b0 ^ inj_bus[4];
else Qvalue_internal <= Dvalue_inj ^ inj_bus[4];
end

assign clk_inj = clk ^ inj_bus[0];
assign rst_n_inj = rst_n ^ inj_bus[1];
assign Dvalue_inj = Dvalue ^ inj_bus[2];
assign Qvalue = Qvalue_internal;

 endmodule
```

(b) Verilog Code for fault instrumented D-Flip Flop

Figure 4.14: Verilog code for D-F/F generated by code generator by reading modified AST from custom parser

4.4. Build Shared Library and Application Binary

Application run on the processor for fault injection simulation is **Dhrystone** benchmark, due to stress on integer operations, which closely represent actual workloads. RISC-V **GNU Tool chain** is used to generate binaries from application program and not **CLang/ LLVM** tool-chain. The first step involves converting dhrystone application in C to a **binary file** to be loaded into to the processor memory by the simulator. A **Makefile** was used for the same and commands in it are described below. They can also be run on a bash command line directly. In the following commands, **gcc** is passed an **-c** option, thus it acts as an **assembler** and assembles the C files to **object files**. Along with dhrystone, libraries **stdlib,syscalls** are also converted to object files(.o). The RISC-V architecture for which the application is assembled is specified by the option **-march**, here as an example, **rv32im** is used.

```
riscv32-unknown-elf-gcc -c -MD -O3 -march=rv32im -DTIME -DRISCV \
-Wno-implicit-int -Wno-implicit-function-declaration dhry_1.c

riscv32-unknown-elf-gcc -c -MD -O3 -march=rv32im -DTIME -DRISCV \
-Wno-implicit-int -Wno-implicit-function-declaration dhry_2.c

riscv32-unknown-elf-gcc -c -MD -O3 -march=rv32im -DTIME -DRISCV stdlib.c
riscv32-unknown-elf-gcc -c -MD -O3 -march=rv32im -DTIME -DRISCV syscalls.c
```

The linker links all the object files to generate a final object file. **Objcopy** converts the object file to a binary. **Objdump** is used to create a disassembly, to aid in debugging.

```
riscv32-unknown-elf-gcc -MD -O3 -march=rv32im -DTIME -DRISCV \
-Wl,-Bstatic,-T,riscv.ld,-Map,dhry.map,--strip-debug -o dhry.elf dhry_1.o \
dhry_2.o stdlib.o syscalls.o -lgcc -lc \

chmod -x dhry.elf
riscv32-unknown-elf-objcopy -O binary dhry.elf  main_app.data
riscv32-unknown-elf-objdump -d -M no-aliases dhry.elf > disass
```

All the above commands can be run by a Makefile by executing following,

```
make dhry
```

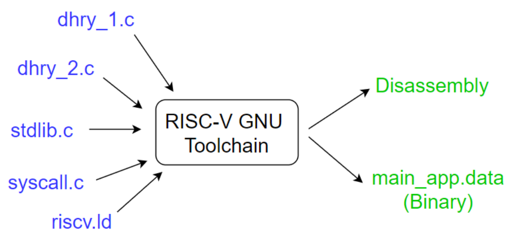


Figure 4.15: Binary and Disassembly generation

Verilator converts the synthesisable Verilog code to a **C++** class. An **Objdir** is created, which has a file **design.h** that is included in test-bench (**tb.cpp**) to instantiate the design.

Instrumented design.v
Verilator
Objdir (design.h)

Figure 4.16: Verilator to transform design to a **c++** class

g++ is used to convert the testbench (including design instantiation) and some verilator libraries to generate a **shared object**, that can be used in a python script. An example g++ command is shown below.

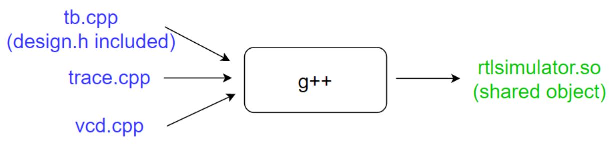


Figure 4.17: Generation of rtl simulator shared object

```
g++ -O0 -shared -fPIC -I/usr/share/verilator/include/ design.cpp \
obj_dir/Vdesign.cpp obj_dir/Vdesign__Syms.cpp obj_dir/Vdesign__Trace.cpp \
obj_dir/Vdesign__Trace__Slow.cpp \
/usr/share/verilator/include/verilated.cpp\
/usr/share/verilator/include/verilated_save.cpp \
/usr/share/verilator/include/verilated_vcd_c.cpp -o rtlsimulator.so
```

4.5. Fault Simulation

The following section describes, the chosen fault model, possible fault classes and finally simulation of the design with fault injection.

4.5.1. Fault models

The fault model used for fault injection was bit-flip, so an **XOR** gate was used to model it in the design. Various possible fault models and the structures to implement them are shown in the table 4.1. **Multiplexers** can actually be used to implement any combinational fault model and to choose between them dynamically at simulation run-time by the user. To implement a delay fault model, a multiplexer alone would not suffice, a delay element such as D-F/F is required. The same custom parser can be used to instrument designs for all the previously discussed fault models by using a different design structure instead of a **XOR** gate. This is one of the potential future works that could be explored for higher fault space coverage of designs.

Design Structure	Fault Model
AND gate with one input tied to ’0’	Stuck at 0
OR gate with one input tied to ’1’	Stuck at 1
XOR gate	Bit Flip
Multiplexer	Open line or indetermination

Table 4.1: Fault models and structures to implement them

4.5.2. Fault classification

The following broad fault classifications are discussed and used in literature,

- Instruction Re-execution [48]
- Instruction Skip [38]
- Abrupt Halt [51]
- Silent Data corruption [34]

Specific instructions could be targeted to compromise the security of the processor. Based on the above, specialised classes of faults for processor were explored and the following were broadly used,

- **Instruction Re-Execution**: The Next PC logic of a processor is effected, forcing the processor to re-fetch an instruction that was just executed and re-execute it.

- **Processor Abrupt Halt**: The flow of execution of instructions is halted. Either the fetch logic, pipeline registers or stall generation logic is corrupted.
- **Early Instruction Termination**: An instruction that take multiple clock cycles to execute is terminated without completion. Multiple sub types of this are possible where, execution flow could be altered and subsequent instructions could be skipped.
- **Delayed Instruction Execution**: Instructions take more clock cycles to finish than expected and their effect on the processor state could be not as intended. Like the previous category, changed execution flow and subsequent instruction skip are possible.
- **Branch Not Taken**: Conditions such as an **If** or **case** could translate to a branch instruction in assembly. To bypass the condition checks, attackers frequently target branch instructions and their conditions are effected. Branches that are supposed to cause a program flow change, thus do not fetch from the target address and continue executing in program order.
- **Branch to different Address**: Address calculation of a branch can be effected and the program can be made to branch to an address where the attacker program sits.
- **Illegal Branch Taken**: Similar to the case of Branch not taken, condition checks are exploited to gain control of the execution flow.
- **Instruction Modified**: Faults could be injected in the **opcode** of the instruction to alter its intended functionality. Register numbers from which data should be read or written to can be modified. The instruction could be converted to effectively function as a **NOP**.
- **PC Changed Not Flow**: A benign error, which just effects the current PC. The subsequent instructions to be fetched are not affected as pipeline registers and Next PC logic are not corrupted.
- **Instruction Skip**: The **Program Counter(PC) + 4** instruction is skipped and **PC + 8** is executed after the current instruction completes execution. Conditional checks can be targeted to exploit this fault.
- **Flow Changed**: The program flow can be altered not just by manipulating and corrupting instructions, but also data in the registers and memory. Data corrupted in a particular clock cycle can be used by the processor in next clock cycles to calculate address for branches or also as conditions for branches.
- **Data Error**: Data alone can be corrupted without changing the program flow. Keys required by cryptographic algorithms like **AES** and **RSA** can be corrupted. Silent data corruption also falls into this category.

  Varying Micro-architectural implementations of a processor define what constitutes a processor state. Based on different implementations, several new fault categories are possible. All **fault classification is done automatically** by the simulation script.

4.5.3. Simulation

On every cycle of fault simulation, the processor state and expected processor state from golden reference model are compared to find mismatches. State machines that keep track of type of fault signature are also updated. Optionally, the processor state is print to simulation log. One clock cycle before simulation ends, summary of all mismatches and final fault signature are printed in the simulation log. This is shown in Figures 4.18 and 4.19.

A snippet of generated simulation log for ’1’ clock cycle is shown in Figure 4.20. If a list of known tests are to be run, a script **test2command.py** can be used to generate commands to be added to the python simulation script, **python_wrapper.py**. The simulation run command requires fields such as fault injection start time, length of fault injection and injection bus indexes of the signals to be injected with fault. These fields are extracted by the script from the test name and commands are saved in a file **runcommands.txt**. These commands are then copied to python_wrapper.py to run the simulation. The following section describes the python simulation script in more detail.

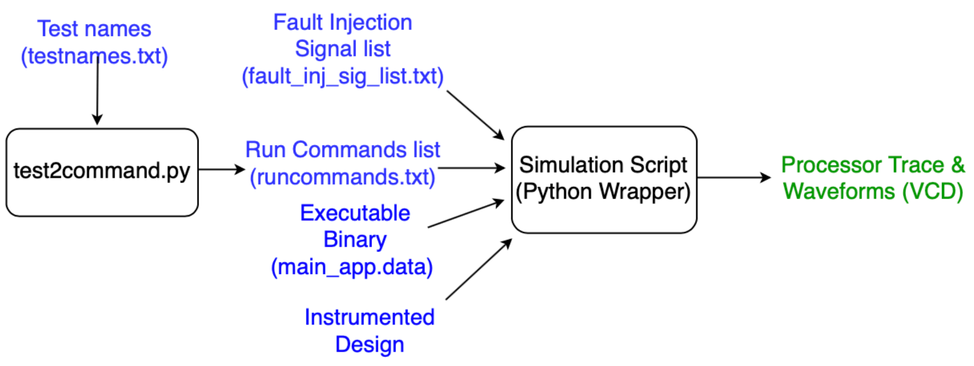


Figure 4.18: Simulation Flow

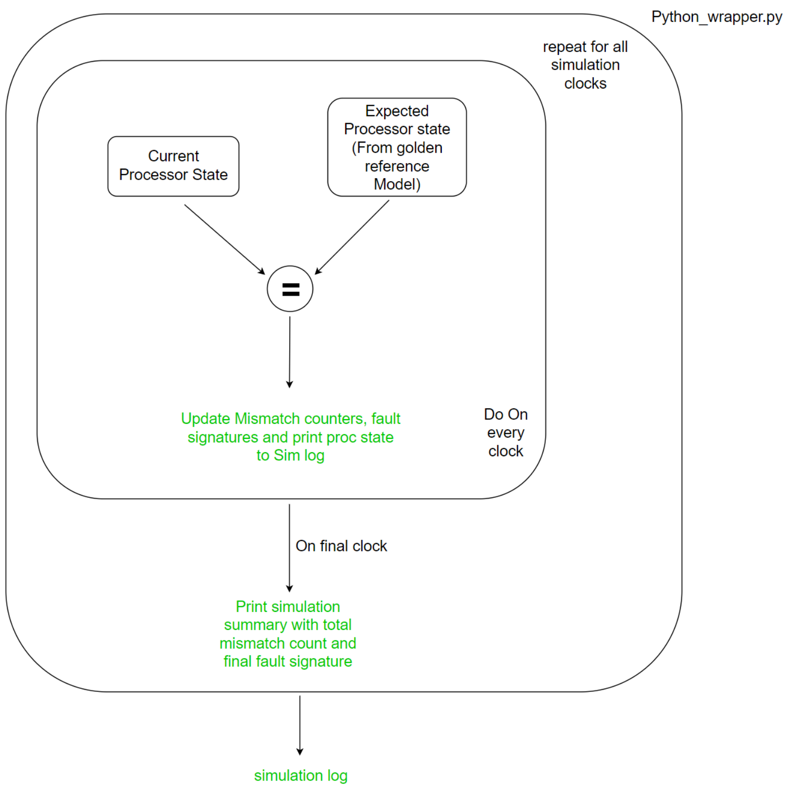
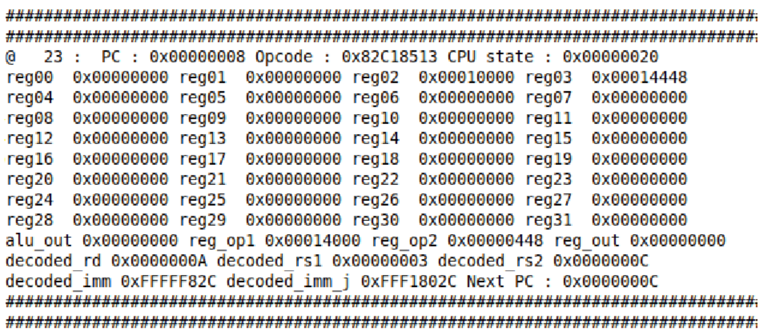


Figure 4.19: Comparison of golden reference state with current processor state



```
###############################################################################
###############################################################################
@   23 :  PC : 0x00000008 Opcode : 0x82C18513 CPU state : 0x00000020
reg00  0x00000000 reg01  0x00000000 reg02  0x00010000 reg03  0x00014448
reg04  0x00000000 reg05  0x00000000 reg06  0x00000000 reg07  0x00000000
reg08  0x00000000 reg09  0x00000000 reg10  0x00000000 reg11  0x00000000
reg12  0x00000000 reg13  0x00000000 reg14  0x00000000 reg15  0x00000000
reg16  0x00000000 reg17  0x00000000 reg18  0x00000000 reg19  0x00000000
reg20  0x00000000 reg21  0x00000000 reg22  0x00000000 reg23  0x00000000
reg24  0x00000000 reg25  0x00000000 reg26  0x00000000 reg27  0x00000000
reg28  0x00000000 reg29  0x00000000 reg30  0x00000000 reg31  0x00000000
alu_out 0x00000000 reg_op1 0x00014000 reg_op2 0x00000448 reg_out 0x00000000
decoded_rd 0x0000000A decoded_rs1 0x00000003 decoded_rs2 0x0000000C
decoded_imm 0xFFFFF82C decoded_imm_j 0xFFF1802C Next PC : 0x0000000C
###############################################################################
###############################################################################
```

Figure 4.20: Processor Trace for ’1’ clock cycle

Python Simulation Script

A python script automates simulation, processor trace and vcd creation. It's algorithm is shown in 2. The previously created **rtlsimulator.so** by **g++** is used as a **DLL/shared library** in **RtlSimulator** class, whose object is created and used in the script. **Ctypes** in python is used to call C foreign functions by wrapping them in pure python. It is used as a DLL in class as shown below,

```
self.sim_dll = CDLL(rtlsimulator.so,mode=RTLD_GLOBAL)
```

Algorithm 2 Python Simulation Script

```
memory[] ← binary file                    // load memory list in python with binary from application
proc_state_list[] ← golden reference file           // read golden reference processor state
sig_list[] ← fault injection signal list file    // read signal names of which faults should be injected
 into

Create RTL Simulator Object(simulator)            // RtlSimulator class loads rtlsimulator.so

Set fault injection period, fault injection simulation time range                  //
Specify whether to generate VCD, Processor Trace                                   //
Set maximum allowed semaphore count                                                //
Specify if it's a golden reference run                                             //

for sig_list[] do                 // iterate over all different signals to be injected with fault
    for data_list[] do   // generated list of data by right shifting '1' for injection bus length
        for final_instr_list do                 // time instances of where faults are to be injected
            simulator.run()          // calls run method, which internally call rtlsim_run in DLL
        end
    end
end
process.join()                                          // Wait for all the process to complete
simulator.close()                                                // Calls rtlsim_close in DLL
```

Algorithm 3 rtlsim_run function

```
Handle memory accesses                                   // Copy contents of binary to python
for Number of clock cycles in a simulation do                                      //
    Read address bus of design and use it to index C++ memory                      //
    Drive data or instruction on memory read bus based on access type              //
    Update C++ memory from memory write bus                                        //
    Save current processor state to an internal variable                           //
    Pass current processor state to python                                         //
    Advance by 1 clock cycle in simulation           // Executing eval() will advance simulation
    Dump to VCD                                                                    //
end
Close VCD Clear C++ memory variable        // memset to clear memory, so as to not effect next
 simulation
```

The **RtlSimulator** class dynamically links **rtlsimulator.so** using **CDLL**. The following are also done in the class,

- Clears fault type counters, **proc_state** mismatch counters, fault state machine variables
- Opens golden reference file.

The test-bench in C++ has two functions rtlsim_run and rtlsim_close, they are called using CDLL from python simulation script. The function rtlsim_run algorithm is explained in 3, it essentially runs the simulation. rtlsim_close function frees up the memory allocated in the test-bench for next simulation. A call back function **cb2()** in rtlsim_run calls **status_callback** in python to send the current processor state to python simulation script. This function calling structure between python and C++ is shown in the Figure 4.21.

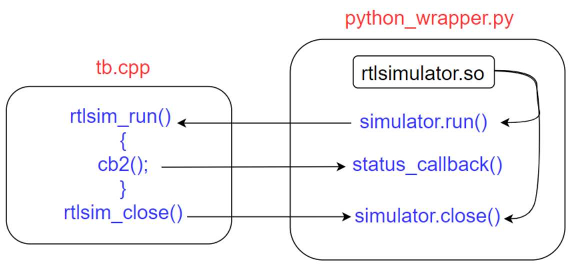


Figure 4.21: Call backs between python simulation script and C++ testbench

A single processor state in python script is maintained as an object of a class **proc_state**. The following constitute a **proc_state**,

- Processor general purpose registers, CPU state register
- Current instruction address & opcode, next PC
- Memory interface signals (access type, valid, ready, read data, write data, address, strobe)
- ALU internal signals (reg_op1, reg_op2, alu_out, reg_out)
- Decoder signals (decoded_rd, decoded_rs1, decoded_rs2, decoded_imm, decoded_immj)

All the above are properties in the class proc_state, a class method to print current state of a processor is also implemented. A list of proc_state for the entire length of simulation with expected processor state is created by running a simulation without any injected faults, called a **golden processor state**. The current processor state and golden processor state are compared on every clock cycle for mismatches. This comparison logic is implemented in **status_callback** in python simulation script, whose algorithm is shown in 4.

Algorithm 4 status_callback function

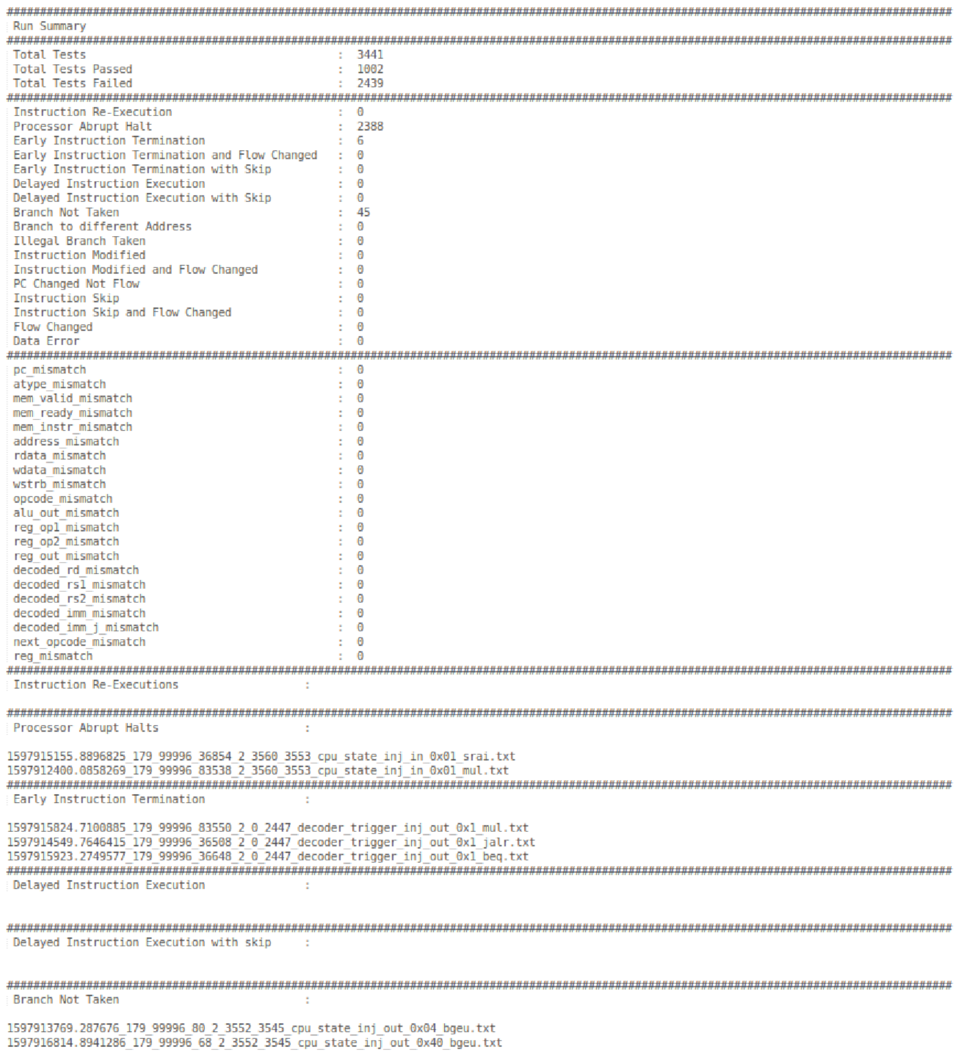
```
Create current proc_state to simulation log file                         // Print processor trace
Read expected golden proc_state from current clock cycle                 //
Update state machines that keep track of specific failure types based on current proc_state   // ex : PC mismatch, Reg mismatch
Compare golden state and current proc_state to update mismatch counters  //
From state machines, determine if a particular fault type is hit         //
Update fault type error signature if any                                 // ex : processor abrupt halt
```

The following instructions are targeted in Dhrystone benchmark for fault injection campaign,

1. ALU instructions - ADDi, SUB, ADD, OR
2. Shifts - Slli, Srai, Srli, Sll, Srl
3. Multiply instructions - Mul, Div, Remu
4. Branchs - Jal, Bgeu, Bltu, Jalr, Bne, Blt, Beq, Auipc
5. Load/ Stores - SW, LW, Lui

4.6. Process Results

The next step is to parse the generated simulation logs and generate the summary report. The script **result_python.py** processes simulation logs to create a run summary. High level summary with total tests run along with failing and passing are mentioned. Number of failures and their file names of a particular fault type are also listed. Finally tests that fail with no particular fault type, but rather with some mismatches are also included. The summary file is named **result.txt** and is displayed by a **top** script after the simulations complete. A sample run summary is shown in Figure 4.22.



```
###########################################################################################################################
 Run Summary
###########################################################################################################################
 Total Tests                                    :  3441
 Total Tests Passed                             :  1002
 Total Tests Failed                             :  2439
###########################################################################################################################
 Instruction Re-Execution                       :  0
 Processor Abrupt Halt                          :  2388
 Early Instruction Termination                  :  6
 Early Instruction Termination and Flow Changed :  0
 Early Instruction Termination with Skip        :  0
 Delayed Instruction Execution                  :  0
 Delayed Instruction Execution with Skip        :  0
 Branch Not Taken                               :  45
 Branch to different Address                    :  0
 Illegal Branch Taken                           :  0
 Instruction Modified                           :  0
 Instruction Modified and Flow Changed          :  0
 PC Changed Not Flow                            :  0
 Instruction Skip                               :  0
 Instruction Skip and Flow Changed              :  0
 Flow Changed                                   :  0
 Data Error                                     :  0
###########################################################################################################################
 pc_mismatch                                    :  0
 atype_mismatch                                 :  0
 mem_valid_mismatch                             :  0
 mem_ready_mismatch                             :  0
 mem_instr_mismatch                             :  0
 address_mismatch                               :  0
 rdata_mismatch                                 :  0
 wdata_mismatch                                 :  0
 wstrb_mismatch                                 :  0
 opcode_mismatch                                :  0
 alu_out_mismatch                               :  0
 reg_op1_mismatch                               :  0
 reg_op2_mismatch                               :  0
 reg_out_mismatch                               :  0
 decoded_rd_mismatch                            :  0
 decoded_rs1_mismatch                           :  0
 decoded_rs2_mismatch                           :  0
 decoded_imm_mismatch                           :  0
 decoded_imm_j_mismatch                         :  0
 next_opcode_mismatch                           :  0
 reg_mismatch                                   :  0
###########################################################################################################################
 Instruction Re-Executions                 :

###########################################################################################################################
 Processor Abrupt Halts                    :

1597915155.8896825_179_99996_36854_2_3560_3553_cpu_state_inj_in_0x01_srai.txt
1597912400.0858269_179_99996_83538_2_3560_3553_cpu_state_inj_in_0x01_mul.txt
###########################################################################################################################
 Early Instruction Termination             :

1597915824.7100885_179_99996_83550_2_0_2447_decoder_trigger_inj_out_0x1_mul.txt
1597914549.7646415_179_99996_36508_2_0_2447_decoder_trigger_inj_out_0x1_jalr.txt
1597915923.2749577_179_99996_36648_2_0_2447_decoder_trigger_inj_out_0x1_beq.txt
###########################################################################################################################
 Delayed Instruction Execution             :


###########################################################################################################################
 Delayed Instruction Execution with skip   :


###########################################################################################################################
 Branch Not Taken                          :

1597913769.287676_179_99996_80_2_3552_3545_cpu_state_inj_out_0x04_bgeu.txt
1597916814.8941286_179_99996_68_2_3552_3545_cpu_state_inj_out_0x40_bgeu.txt
```

Figure 4.22: Run Summary

4.7. Code Profiling and Optimisations

As the fault injection simulation campaigns need to run tests in the order of 100k, execution time of each simulation is very important. One of the ways to improve run time is to profile the code to identify bottlenecks. To profile the python code, Cprofile, line_profiler and pprofile were used. **Cprofile** gives the total runtime, along with time spent in executing each function and number of times the function was called. This helps to identify and optimise the function that consumes the most time to execute. Reducing the function call count is another way to improve the execution time. **Cprofile** can invoked while running the script using the following command,

```
$ python3 -m cProfile -s tottime python_wrapper.py
```

The following is a snippet from the log generated by Cprofile for a simulation with no fault injections. The length of the simulation was 49998 clock cycles. As it can be seen, most of the time in simulation is consumed by run, status_callback methods. The run method is where the actual simulation is done. The status_callback is called 49998 times are it is invoked every clock cycle. The ’module’ is actually where reading of the application binary and setting internal variables of the script takes place. When a regression is run, the ’module’ part is only run once, but run and status_callback invoked by every simulation. So, the effective total simulation time is 3.078 seconds.

```
 807622 function calls (807263 primitive calls) in 4.611 seconds

Ordered by: internal time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    1.598    1.598    3.143    3.143 python_wrapper.py:1027(run)
 49998    1.480    0.000    1.543    0.000 python_wrapper.py:288(status_callback)
     1    1.084    1.084    4.611    4.611 python_wrapper.py:1(<module>)
     1    0.138    0.138    0.138    0.138 python_wrapper.py:1021(close)
 50001    0.119    0.000    0.119    0.000 {method 'split' of 'str' objects}
 99998    0.070    0.000    0.070    0.000 python_wrapper.py:50(__init__)
214952    0.022    0.000    0.022    0.000 {method 'append' of 'list' objects}
 81013    0.016    0.000    0.016    0.000 {method 'read' of '_io.BufferedReader' objects}
 81013    0.011    0.000    0.011    0.000 {method 'hex' of 'bytes' objects}
 46347    0.010    0.000    0.010    0.000 {built-in method builtins.bin}
 73172    0.009    0.000    0.009    0.000 {method 'popleft' of 'collections.deque' objects}
 73172    0.008    0.000    0.008    0.000 {method 'append' of 'collections.deque' objects}
 13242    0.004    0.000    0.004    0.000 {built-in method builtins.isinstance}
     6    0.003    0.001    0.003    0.001 {built-in method _imp.create_dynamic}
    29    0.003    0.000    0.003    0.000 {built-in method marshal.loads}
   107    0.003    0.000    0.006    0.000 <frozen importlib._bootstrap_external>:1233(find_spec)
  2090    0.003    0.000    0.003    0.000 {built-in method _codecs.utf_8_decode}
```

line_profiler lists time taken by individual lines of code. This was used further to narrow down the search for lines that slow down a particular functions. **@profile** decorator should be added to functions that are to be profiled. It can be executed by running the following,

```
$ kernprof -l -v python_wrapper.py
```

The following is a snippet of the log generated by line_profiler for a simulation with no faults injected. As can be seen, time taken per line and number of times the line is executed in a function along with their line number are specified.

```
Timer unit: 1e-06 s

Total time: 0.257298 s
File: python_wrapper.py
Function: __init__ at line 49

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    49                                           @profile
    50                                           def __init__(self, param_list):
    51
    52     99998     138351.0      1.4     53.8      self.reg0_addr = param_list[22:54]
    53     99998     118947.0      1.2     46.2      self.state_variables = param_list[0:22]

Total time: 0.00017 s
File: python_wrapper.py
Function: check_status at line 235
```

pprofile, which was developed inspired from line_profiler was also used. pprofile also does profiling at line level granularity and is also thread aware. The profiler gives hits, time per hit, total time and percentage of total simulation time per every line in the script by default. Modification of source to enable profiling like in the case of **line_profiler** is not required. Also, recursive methods profiling is handled efficiently compared to in line_profiler. The profiling of libraries and imported modules used is also shown. A snippet is shown below,

```
Command line: python_wrapper.py
Total duration: 108.105s
File: python_wrapper.py
File duration: 107.88s (99.79%)
Line #|      Hits|         Time| Time per hit|      %|Source code
------+----------+-------------+-------------+-------+-----------
     1|         2|  0.000343084|  0.000171542|  0.00%|from ctypes import *
(call)|         1|    0.0366392|    0.0366392|  0.03%|# <frozen importlib._bootstrap>:966 _find_and_load
(call)|         1|  0.000100613|  0.000100613|  0.00%|# <frozen importlib._bootstrap>:997 _handle_fromlist
     2|         1|  4.24385e-05|  4.24385e-05|  0.00%|import math
     3|         1|  6.12736e-05|  6.12736e-05|  0.00%|import time
     4|         1|  5.38826e-05|  5.38826e-05|  0.00%|import re
     5|         1|  0.000143051|  0.000143051|  0.00%|import csv
(call)|         1|    0.0132754|    0.0132754|  0.01%|# <frozen importlib._bootstrap>:966 _find_and_load
     6|         1|  0.000141621|  0.000141621|  0.00%|from collections import deque
(call)|         1|  0.000107288|  0.000107288|  0.00%|# <frozen importlib._bootstrap>:997 _handle_fromlist
     7|         1|  5.53131e-05|  5.53131e-05|  0.00%|import multiprocessing
     8|         0|            0|            0|  0.00%|
     9|         0|            0|            0|  0.00%|
    10|         1|  5.26905e-05|  5.26905e-05|  0.00%|def search_signal_in_file (file_name,signal_name):
    11|         0|            0|            0|  0.00%| text_file = open(file_name,"r")
    12|         0|            0|            0|  0.00%| msb_lsb_tuple_list = []
    13|         0|            0|            0|  0.00%| signal_name_inj = signal_name + "_inj"
```

Based on the above tool logs, the following optimisation were made in the code,

- Unnecessary number conversions to integer type, by calling **int()** were removed, inorder to reduce the total number of function calls.
- Replaced variables by lists, where possible.
- Initialising lists by **'*'** instead of **range()** function.
- Reading from **ctypes** arrays by **':'**, instead of **range()**.
- Removed many unnecessary internal variables and other generic code optimisations.

A code version with only **Numpy** arrays instead of python lists was implemented to achieve better performance. Instead, it performed worse than using normal lists. Numpy arrays offer performance gain only when the array size is large. In the simulation script, the array sizes are not large enough to cancel out effect of loading numpy and initialising it's arrays.

PyPy is an alternative python implementation, that uses just-in-time compilation to reduce execution times. PyPy was also used for python simulation script to improve the execution time. The performance obtained is worse compared with original python implementatio **CPython** and minutely beter than the numpy version.

An option to generate all simulation logs in **.csv** was added to the script, to enable importing into a database inorder to support **SQL** queries. **SQL** queries can be used for more advanced processing of log files, such as to obtain information about instructions executed right before simulation fail and group failures based on them. Reading and writing golden reference machine state file is slower if it's in **.csv** format, so **.txt** format is still used. The following are run-times of a single simulation with different optimisations,

- **Original** - 6.556 seconds
- **Numpy** - 7.750 seconds
- **PyPy** - 7.367 seconds
- **Optimised** - 3.558 seconds

4.8. Multiprocessor Support

Code profiling and subsequent optimisations improve simulation performance for single core. To further reduce execution time for fault injection simulation campaigns, multiprocessors should be used. Python provides two classes for multiprocessing, namely **process** and **pool**. Both the approaches are evaluated for performance. An option to enable and disable multiprocessing support is provided in the python simulation script.

Pool

If data parallelism is required, Pool based implementation is to be used. The same function is executed with multiple sets of input data. Input data is distributed across multiple processes. The simulation script worked for 2-processes on a 4-core system and crashes any higher number of processes. Simulation time of **2.14** seconds is obtained with 2-processes pool. An algorithmic level python code implementation of pool is shown below,

```
def pool_func (pool_params):
    simulator.run(pool_params)

pool__params_list = []

pool = multiprocessing.Pool(2) #3 Crashes the system

for sig_list[] : #For every signal in the fault_inj_sig_list.txt
    for shifted_data_list[] : #Generate list of data by right shifting '1'
        for final_instr_list[] : #Various simulation time instances to target multiple instructions
            multiproc_params_list.append #Generating multiple sets of input data

pool.map(multiproc_func,multiproc_params_list) #runs simulator.run()
```

Process

For a function based parallelism, where for same data, multiple functions are to be run, **process** is used. The number of processes is controlled by number of semaphores. A different process is allocated for each signal to be injected with fault. Algorithmic python code for process implementation is shown below,

```
def proc_func (individual_signal,sema):
    for shifted_data_list[] : #Generate list of data by right shifting '1'
        for final_instr_list[] : #Various simulation time instances to target multiple instructions
            simulator.run
pool__params_list = []

sema = multiprocessing.Semaphore(2)

all_processes = []

for sig_list[] : #For every signal in the fault_inj_sig_list.txt
    sema.acquire()
    p = multiprocessing.Process(target=multiproc_func,args=(individual_signal,sema))
    all_processes.append(p)
    p.start()

for p in all_processes:
p.join()
```

With 2 processes each simulation took **1.7 seconds**. Having 4 processes was slower and 20 processes fails with an error, **too many files open**. The **Pool** implementation of parallelism is slower by **26%** compared to **Process** implementation and thus was chosen for fault injection campaigns. This is unexpected given, what **Pool** and **Process** are meant to be used for. This could be attributed to the fact that, a **4-core** system with only **2-cores** available to **virtual box** is used, with high system memory constraints. If processes are spawned on a computer cluster designed to handle parallelism, the performance of **Pool** should easily outpace that of **Process** for the python simulation script.

5
Case study : FI of PicoRV32, DarkRISCV

5.1. PicoRV32

PicoRV32 is a highly configurable processor, it was design to be chosen either to be high performance or low area and power as explained in section 3.2.1. As the objective here is fault injection, design with all extensions and features is preferred, so the high performance variant **RV32IM** core is chosen. To configure the core, following Verilog parameters are to be set,

- **ENABLE_REGS_16_31**: This enables registers 16 to 31 in design, giving processor more register to work with. If they are not enabled, it corresponds to **RV32E** core configuration.
- **ENABLE_REGS_DUALPORT**: Two read ports instead of one, improves the performance.
- **BARREL_SHIFTER**: By default, shift is done in two stages. A barrel shifter completes a shift operation in ’1’ clock cycle like any ALU operation.
- **ENABLE_PCPI**: The Co-Processor interface is used to implement external cores.
- **ENABLE_FAST_MUL**: A normal multiplier completes **MUL** instruction and **MULH** instruction in 40 and 72 cycles respectively. The fast multiplier is a single cycle multiplier and it uses the PCPI interface.
- **ENABLE_DIV**: A divider that executes **DIV/REM** in 40 clocks is enabled.

Processor State

The following constitute a processor state in PicoRV32. They are compared with expected processor state from golden reference model on every clock cycle.

- General purpose registers - 0 to 31.
- Current Program Counter, Opcode and CPU state register.
- Temporary ALU result (ALU_Out), ALU operands (reg_op1,reg_op2).
- Memory interface signals - valid, ready, read/write data, address, strobe and access type.
- Instruction operands’ decoder signals - rs1 (1st source operand), rs2 (2st source operand), rd (destination operand), immediate.

5.1.1. Fault Injection & Fault Bucketing

For fault injection, **95 important signals and states** were chosen from a total of 213. Data bus signals such as **mem_wdata** and **mem_rdata** were not included as the focus was control errors and not data errors. Many signals and register such as **pcpi_div_outsign** , **pcpi_fast_mul_rd_q** that are internal to divider and multiplier were ignored as they effect only divide and multiply instructions, while drastically increasing the required number of simulations. The chosen signals fall into the following categories,

- **Decoder** - 30
- **Look ahead interface** - 7
- **Co-Processor interface** - 5
- **Main State Memory and ALU** - 27
- **Memory** - 17

The following are some note worthy points about fault injection simulation in PicoRV32,

- The **injection bus width is 5715**, it is added as a top level input port to the instrumented design.
- As defined on page 42, ALU instructions, shifts, Multiply/Divide, branches and Load/Store are targeted for fault injection in dhrystone.
- The times at which faults are injected are once every clock cycle from **2-cycles** before the instruction executes to **1-cycle** after it completes. This is to make sure that, **faults in fetch,decode and memory write back** signals actually effect the intended instruction.
- A **fault campaign involving 99534** simulations was run for PicoRV32.
- A total of **17 different failure signatures** are possible. Signatures are similar to classes mentioned in page 37.

5.1.2. Results

In the total **99534** simulations, **85093** pass without any error and **14441** simulations fail. The corresponding percentages are shown in the Figure 5.1. A test is classified as being passed, if for all the length of simulation, the program flow and processor state match with golden reference model. It is classified as fail otherwise.

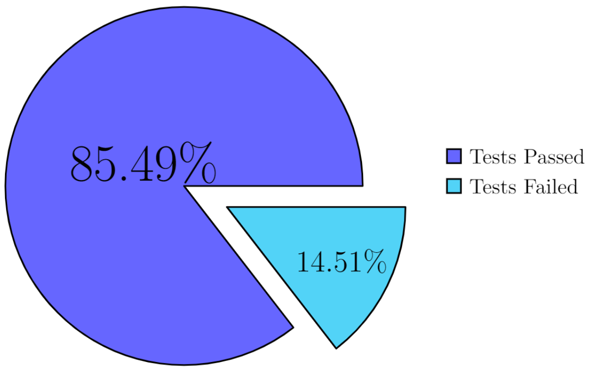


Figure 5.1: Fault campaign pass and fail percentages for PicoRV32

Of the total **14441** fails, the distribution of the different fault signatures is as shown in Figure 5.2. Mismatches that don't cause either a change to control flow or data are grouped under **others** category.

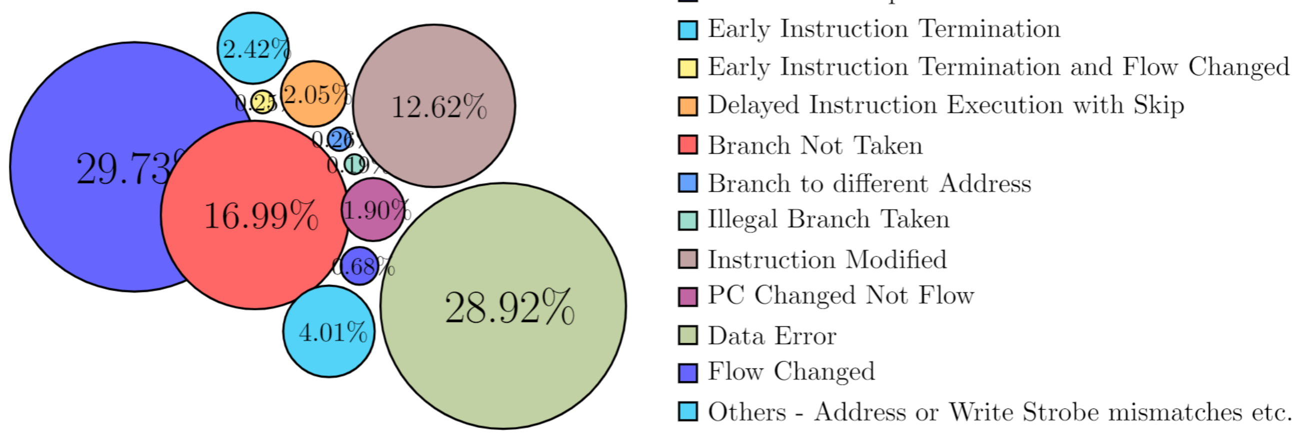


Figure 5.2: Failure distribution for PicoRV32

These are transient changes in signals/ registers that are corrected in the subsequent clock cycle without having any effect on the processor.

As seen above, most of the injected faults don't translate to an error in the processor state or affect the program flow. Ignoring data errors, which are essentially **silent data corruptions** as they don't effect the program flow. Of the remaining fails, **Processor abrupt halt, Branch not take and Instruction modified** constitute the bulk of fails around **60%**. Knowledge about such failures can help processor designers to make choices about choosing countermeasure for particular modules in processors inorder to yield high returns. PicoRV32 doesn't implement a **watchdog timer or duplication of decode logic**, which could have helped in processor abrupt halt and instruction modified cases respectively.

5.2. DarkRISCV

The DarkRISCV processor was designed to replace high performing Coldfire processors running at more than 75MHz and to be implemented on spartan-6 FPGAs. As such, it too is highly configurable like PicoRV32. Pyverilog can't handle **'ifdef** constructs and takes the **if** part by default, ignoring the **'defines** in the configuration file (**config.vh**). For PicoRV32, the design was manually edited, which is error prone. In order to automate the process of manually configuring the Verilog parameters and inturn editing Verilog design file, **yosys** can be used. Initially, parameters to be configured are **'defined** in **config.vh**. Then running the following on command line gives the configured design.

```
echo ”read_verilog -ppdump darkriscv.v” > synth.ys
yosys synth.ys > darkriscv_yosys.v
```

The following parameters are selected in the design,

- **__3STAGE__**: Enables 3 stage pipeline
- **__MAC16X16__**: To add MAC block to the design.
- **__HARVARD__**: Set to '0' as VonNeumann architecture is desired.

Threading and performance measurement are also disabled. DarkRISCV currently doesn't have support for multiply or divide instructions, as such it implements only **RV32E**. The parameters were chosen so that, it resembles PicoRV32 as much as possibles, with an aim of making fault injection comparison of the two processors fair.

Processor State

DarkRISCV doesn't have a CPU state register or status register. It also has only 15 general purpose registers as **RV32E** ISA is chosen. The three stages of the pipeline have a PC corresponding to them, to indicate the instruction in that stage.

- General purpose registers - 0 to 15.
- Current Program Counter (PC), next PC, next PC2 and Opcode.
- Outputs from various modules in ALU - LData, KData and RMData.
- Memory interface signals - read/write data and address.

Code adoption

As DarkRISCV is a pipelined design, the state machines for fault classification used in **python_wrapper.py** (simulation script), are not valid as they were written with PicoRV32, which is a single cycle processor in consideration. This is also impacted by change in variables that constitute a processor state. The state machines were rewritten to reflect these changes.

5.2.1. Fault Injection & Fault Bucketing

A total of **39 signals and registers** were chosen for fault injection. Unlike PicoRV32, where faults only targeted control signals, here all control signals and internal data outs from ALU modules were selected to increase the simulation count. The injection bus is of width **1317**. The signals chosen fall into these groups,

- **Control Signals** - FLUSH, FCT3, FCT7, OPCODE, HLT : **5**
- **Stage 1 Pipeline signals (Decode)** - XLUI, XMAC, XRCC, etc. : **10**
- **Stage 1 Pipeline signals (Execute)** - LCC, SCC, MCC, etc. : **10**
- **Register file control** - RESMODE, BE, S1PTR, S2PTR, DPTR : **5**
- **Jump Control** - BMX, JAL, JALR, JVAL : **4**
- **ALU outputs** - LDATA, SDATA, RMDATA, KDATA : **4**

The fault campaign had **31850** simulation for DarkRISCV. **18 failure signatures** are possible. Compared to PicoRV32 the only new signature is that of **default error**, as the name implies is hit if none others are. The following 2 categories of mismatches are also added as the architecture is different compared to PicoRV32,

- Next PC mismatch, Next PC2 mismatch.
- Data address, Data read/write mismatch.

5.2.2. Results

A total of **10324** simulations passes and **21526** simulations fail of the total **31850** simulations. The respective percentages are shown in the Figure 5.3.

Various fault signature as a percentage of total fails are shown in Figure 5.4. A majority of the faults are data errors, which is expected as faults are injected into 4 ALU output signals. As the focus is on control errors, ignoring data errors, the highest errors are branch to a different address, Branch not taken and PC changed. In PC changed fault, the PC value itself is corrupted by doesn't effect the next instruction in the pipeline or current instruction as it is overwritten in the next clock cycle. Thus countermeasures for branching logic can avoid **12.78%** simulation fails. In-order to fix the major data errors, duplication technique such as **Triple modular redundancy** can be implemented.

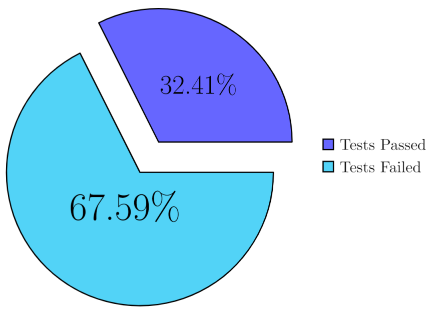


Figure 5.3: Pass/Fail percentages for DarkRISCV

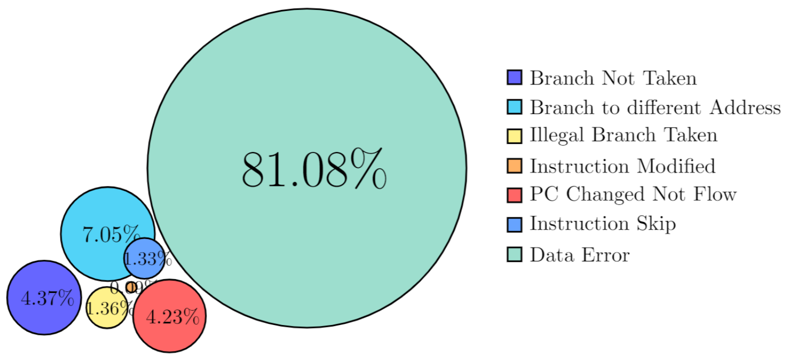


Figure 5.4: Failure Signatures for DarkRISCV

5.3. Comparison of processors

As both PicoRV32 and DarkRISCV both implement the RISC-V ISA, analysing their types of failures can be used to draw insights to RISC-V ISA and implemented micro-architecture vulnerabilities.

Processor Abrupt Halt

Approximately, **30%** of the failures are from this category in case of PicoRV32 and none in DarkRISCV. Most of the fails in PicoRV32 are due to direct or indirect fault into **mem_do_prefetch** signal, which controls the **cpu_state** updation to **fetch state**. As can be seen from the following code, **decoder_trigger** is also effected and thus not only new instruction is not fetched, the current instruction that is already fetched is not executed.

```
if (!mem_do_prefetch && mem_done) begin
    cpu_state <= cpu_state_fetch;
    decoder_trigger <= 1;
    decoder_pseudo_trigger <= 1;
end
```

Where as in DarkRISCV, there is no equivalent signal to **mem_do_prefetch** in the implementation. The **HALT** signal is an input to the processor and thus, even if a fault is injected on it for '1' clock cycle, it recovers in the next. This prevents the processor to go into a complete **HALT**. The PCs are updated in DarkRISCV based on **reset and HALT** as shown in the code below,

```
input HLT;                          // halt
PC    <= HLT ? PC : NXPC;           // current program counter
NXPC  <= HLT ? NXPC : NXPC2
NXPC2 <=  RES ? 0 : HLT ? NXPC2 :   // reset and halt
                    JREQ ? JVAL :   // jmp/bra
                    NXPC2+4;        // normal flow;
```

Branch not taken & Branch to a different address

Branch not taken is the 2nd and 3rd highest signatures for PicorRV32 and DarkRISCV respectively. In PicorRV32, reg **latched_branch** is used in control logic to keep track of branch and jump instructions. When a fault is injected in it, a branch can be made to be not taken. In DarkRISCV case, **JAL, JALR, JREQ, BMUX** regs in control logic are used to keep track of keep track of branches and registers.

```
        assign next_pc = latched_store && latched_branch ? reg_out & ~1 : reg_next_pc;
    case (1):
        latched_branch: begin
        cpuregs_wrdata = reg_pc + (latched_compr ? 2 : 4);
        cpuregs_write = 1;
        end
```

Branch to a different address contributes to only **38** simulation fails in PicoRV32 , whereas it is the 2nd highest signature in DarkRISCV simulations. The difference arises since there is no dedicated reg for holding jump address value in PicoRV32, whereas DarkRISCV has the reg **JVAL**. JVAL is used to drive NXPC2 and NXPC, which are PC address for 1st two pipeline stages. A fault in this directly translates to a corrupted branch address.

```
        NXPC2 <=  RES ? 0 : HLT ? NXPC2 :    // reset and halt
                            JREQ ? JVAL :    // jmp/bra
                            NXPC2+4;         // normal flow
        NXPC <= RES ? 0 : HLT ? NXPC :       // reset and halt
                          JREQ ? JVAL :      // jmp/bra
                          NXPC+4;            // normal flow
```

Instruction Modified

Its the 3rd highest signature for PicorRV32 and causes only **20** fails in DarkRISCV. In PicorRV32, **current_pc** is a register, driven by **reg_next_pc**, alu_out_q and reg_out. Injecting a fault into reg_next_pc on any cycle the instruction is being executed, corrupts current_pc.

```
    current_pc = reg_next_pc;
    case (1'b1)
        latched_branch: begin
            current_pc = latched_store ? (latched_stalu ? alu_out_q : reg_out) & ~1 : reg_next_pc;
        end
```

On the other hand, PC, NXPC and NXPC2 feed into each other conditionally. Injecting a fault into either of them doesn't translate to a failing test as instruction decode control signals are latched for every pipeline stage.

Delayed Instruction Termination

These types of failures are not seen in DarkRISCV as it is a 3-stage pipeline and if an instruction in pipeline is corrupted, it is still completed in the same clock cycle without halting the pipeline. A quarter of a percentage fails are seen in PicoRV32 with this signature. The instructions get delayed by one or more clock cycles and may also have a corrupted output. This is possible in PicoRV32 only because the design is not pipelined.

PC changed not flow

With **911** and **274** simulation failing in DarkRISCV and PicoRV32, it is a significant fail in the former but not the latter. This can be attributed to the fact that DarkRISCV has 3 registers to hold PCs, **PC, NXPC and NXPC2** that drive each other, where as PicoRV32 has **current_pc**, which is to be corrupted at a specific clock cycle to not change the flow. So, corrupting the PC without effecting flow is easier in DarkRISCV.

Instruction Skip

No cases of **instruction skip** were seen in PicoRV32. There are few cases of **delayed instruction with skip**, where the decode logic doesn't take into account the updated processor state and re-fetches the next instruction. Injecting faults into **JVAL** for some address can cause and instruction skip in DarkRISCV.

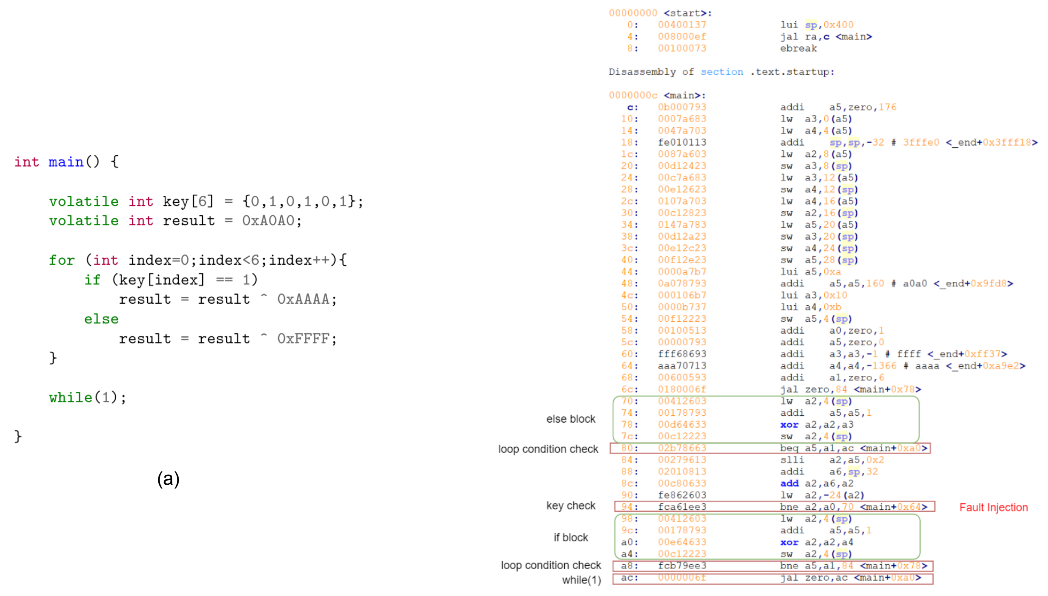
An interesting finding is, none of the injections in any signal was able to produce an **instruction re-execution** in either of the processors. This type of fault requires multiple signals and registers to be corrupted spread over few clock cycles. It can be inferred from the analysis that, a pipeline design is less likely to suffer an abrupt halt than an single cycle design. Having registers in control logic with a well defined functionality is very vulnerable to attack. Attackers don't have access to design code, thus their implementation by design tool will determine it's vulnerability.

5.4. Implementation of Countermeasures

Inverse operation check as a software countermeasure for branch not taken fault and Watchdog timer as a hardware countermeasure for processor abrupt halt fault signature in PicoRV32 are discussed in this section.

5.4.1. Complementary Double check implementation

Branch not taken failure signature was the second highest in both PicoRV32 and DarkRISCV. The program in Figure 5.5a, which has a for loop with an if condition inside, controlled by binary bits similar to Modexp operation in Figure 2.21 is used as a test program. A fault is injected with and without countermeasure, to test its effectiveness. The equivalent assembly code for the test program is shown in Figure 5.5b. A Branch not taken fault is induced, when the **bne** instruction indicated in the Figure 5.5a is executing. This corresponds to an **if** condition, that checks the key value in the C test program. The expected value of result is **0xF5F5**, but since the **if** condition for index 4 is skipped, the resulting value is **0xA0A0**.


```c
int main() {

    volatile int key[6] = {0,1,0,1,0,1};
    volatile int result = 0xA0A0;

    for (int index=0;index<6;index++){
        if (key[index] == 1)
            result = result ^ 0xAAAA;
        else
            result = result ^ 0xFFFF;
    }

    while(1);

}
```

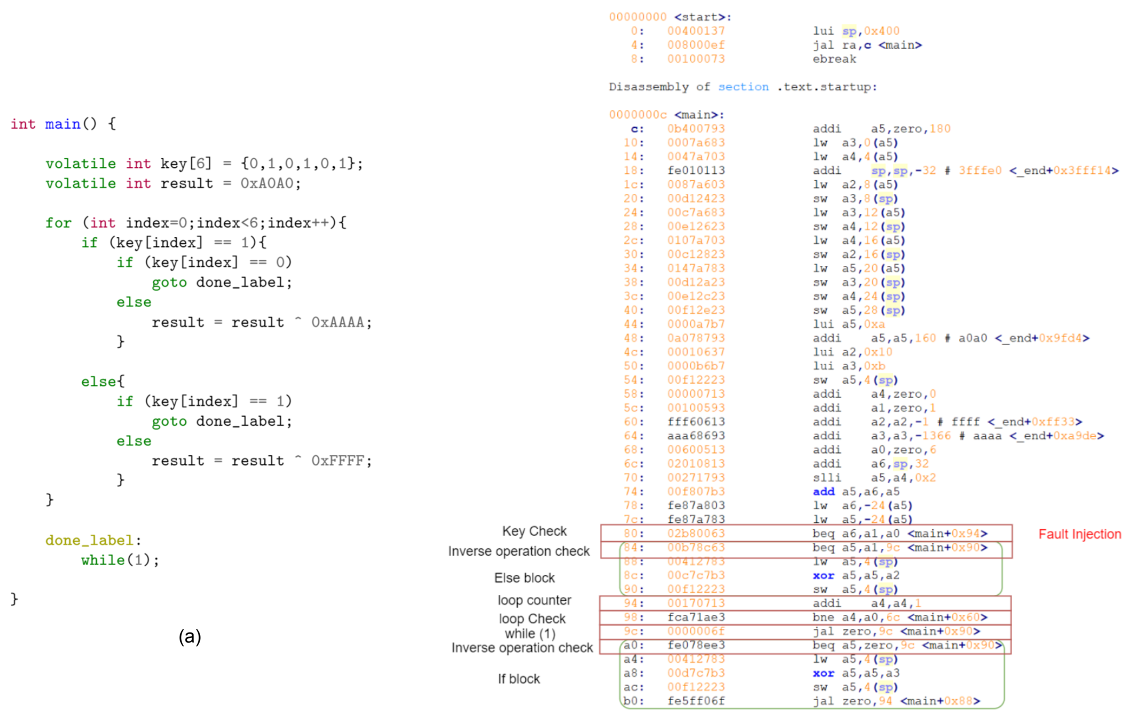
(a)

```
00000000 <start>:
   0:   00400137          lui sp,0x400
   4:   008000ef          jal ra,c <main>
   8:   00100073          ebreak

Disassembly of section .text.startup:

0000000c <main>:
   c:   0b000793          addi    a5,zero,176
  10:   0007a683          lw  a3,0(a5)
  14:   0047a703          lw  a4,4(a5)
  18:   fe010113          addi    sp,sp,-32 # 3fffe0 <_end+0x3fff18>
  1c:   0087a603          lw  a2,8(a5)
  20:   00d12423          sw  a3,8(sp)
  24:   00c7a683          lw  a3,12(a5)
  28:   00e12623          sw  a4,12(sp)
  2c:   0107a703          lw  a4,16(a5)
  30:   00c12823          sw  a2,16(sp)
  34:   0147a783          lw  a5,20(a5)
  38:   00d12a23          sw  a3,20(sp)
  3c:   00e12c23          sw  a4,24(sp)
  40:   00f12e23          sw  a5,28(sp)
  44:   0000a7b7          lui a5,0xa
  48:   0a078793          addi    a5,a5,160 # a0a0 <_end+0x9fd8>
  4c:   000106b7          lui a3,0x10
  50:   0000b737          lui a4,0xb
  54:   00f12223          sw  a5,4(sp)
  58:   00100513          addi    a0,zero,1
  5c:   00000793          addi    a5,zero,0
  60:   fff68693          addi    a3,a3,-1 # ffff <_end+0xff37>
  64:   aaa70713          addi    a4,a4,-1366 # aaaa <_end+0xa9e2>
  68:   00600593          addi    a1,zero,6
  6c:   0180006f          jal zero,84 <main+0x78>
  70:   00412603          lw  a2,4(sp)
  74:   00178793          addi    a5,a5,1
  78:   00d64633          xor a2,a2,a3
  7c:   00c12223          sw  a2,4(sp)
  80:   02b78663          beq a5,a1,ac <main+0xa0>
  84:   00279613          slli    a2,a5,0x2
  88:   02010813          addi    a6,sp,32
  8c:   00c80633          add a2,a6,a2
  90:   fe862603          lw  a2,-24(a2)
  94:   fca61ee3          bne a2,a0,70 <main+0x64>
  98:   00412603          lw  a2,4(sp)
  9c:   00178793          addi    a5,a5,1
  a0:   00e64633          xor a2,a2,a4
  a4:   00c12223          sw  a2,4(sp)
  a8:   fcb79ee3          bne a5,a1,84 <main+0x78>
  ac:   0000006f          jal zero,ac <main+0xa0>
```

(b)

Figure 5.5: Test program and its assembly code

When inverse operation check is added to the test program as a countermeasure, though one branch is skipped by the injected fault, the 2nd branch does the complementary check and catches the fault. It then redirects the test to outside the loop. this is shown in the Figure 5.6. The test program was run on PicoRV32 and the fault was injected using the python simulation framework. In order to activate the branch not taken fault signature, fault is injected into **decode_rs2** signal to make it to pick a wrong operand for **bne** instruction.



```
int main() {

    volatile int key[6] = {0,1,0,1,0,1};
    volatile int result = 0xA0A0;

    for (int index=0;index<6;index++){
        if (key[index] == 1){
            if (key[index] == 0)
                goto done_label;
            else
                result = result ^ 0xAAAA;
            }

        else{
            if (key[index] == 1)
                goto done_label;
            else
                result = result ^ 0xFFFF;
            }
    }

    done_label:
        while(1);

}
```

(a)

```
00000000 <start>:
   0:   00400137            lui sp,0x400
   4:   008000ef            jal ra,c <main>
   8:   00100073            ebreak

Disassembly of section .text.startup:

0000000c <main>:
   c:   0b400793            addi    a5,zero,180
  10:   0007a683            lw  a3,0(a5)
  14:   0047a703            lw  a4,4(a5)
  18:   fe010113            addi    sp,sp,-32 # 3fffe0 <_end+0x3fff14>
  1c:   0087a603            lw  a2,8(a5)
  20:   00d12423            sw  a3,8(sp)
  24:   00c7a683            lw  a3,12(a5)
  28:   00e12623            sw  a4,12(sp)
  2c:   0107a703            lw  a4,16(a5)
  30:   00c12823            sw  a2,16(sp)
  34:   0147a783            lw  a5,20(a5)
  38:   00d12a23            sw  a3,20(sp)
  3c:   00e12c23            sw  a4,24(sp)
  40:   00f12e23            sw  a5,28(sp)
  44:   0000a7b7            lui a5,0xa
  48:   0a078793            addi    a5,a5,160 # a0a0 <_end+0x9fd4>
  4c:   00010637            lui a2,0x10
  50:   0000b6b7            lui a3,0xb
  54:   00f12223            sw  a5,4(sp)
  58:   00000713            addi    a4,zero,0
  5c:   00100593            addi    a1,zero,1
  60:   fff60613            addi    a2,a2,-1 # ffff <_end+0xff33>
  64:   aaa68693            addi    a3,a3,-1366 # aaaa <_end+0xa9de>
  68:   00600513            addi    a0,zero,6
  6c:   02010813            addi    a6,sp,32
  70:   00271793            slli    a5,a4,0x2
  74:   00f807b3            add a5,a6,a5
  78:   fe87a803            lw  a6,-24(a5)
  7c:   fe87a783            lw  a5,-24(a5)
  80:   02b80063            beq a6,a1,a0 <main+0x94>
  84:   00b78c63            beq a5,a1,9c <main+0x90>
  88:   00412783            lw  a5,4(sp)
  8c:   00c7c7b3            xor a5,a5,a2
  90:   00f12223            sw  a5,4(sp)
  94:   00170713            addi    a4,a4,1
  98:   fca71ae3            bne a4,a0,6c <main+0x60>
  9c:   0000006f            jal zero,9c <main+0x90>
  a0:   fe078ee3            beq a5,zero,9c <main+0x90>
  a4:   00412783            lw  a5,4(sp)
  a8:   00d7c7b3            xor a5,a5,a3
  ac:   00f12223            sw  a5,4(sp)
  b0:   fe5ff06f            jal zero,94 <main+0x88>
```

(b)

Figure 5.6: Test program and assembly code with countermeasure

5.4.2. Watchdog Timer

Around **4300** simulations fail with processor abrupt halt failure signature in fault injection campaign for PicoRV32, that accounts for **30%** of the total failures. A watchdog timer, that **restarts the processor after 80 clocks of inactivity** was added to PicoRV32 as a countermeasure for processor abrupt halt fail signature. To finally test the implementation, 10K simulations, were selected from the total 100k simulations uniformly to include all failure signatures and run. This makes sure that the watchdog while preventing processor abrupt halt fails, doesn't effect functionality of other modules. Multiple implementations of watchdog were evaluated for highest simulation pass percentage. The watchdog resets the processor only after **80** clock cycles because **MULH** instruction takes **72** clock cycles to complete. To recover the processor to it's previous state once it is halted, following several approaches were chosen based on the new cpu state they put the processor in,

- **CPU state based on current instruction**: Based on the type of current instruction being executed before halt, the cpu state on watchdog reset is determined. For shift instructions, cpu state is set to decode the operands again. Whereas for store and loads, the processor state is set to refetch from memory and for other class of instructions, the instruction is refetch. These states were determined studying the processor implementation of these instructions. A pass percentage of **39** was obtained by this approach.
- **Set CPU state always to Re-fetch**: In this approach, the halted instruction is always refetched while also clearing temporary registers in ALU like **alu_out**, **reg_op1**, **reg_op2** that hold ALU result and ALU operand 1 and 2. **56%** pass rate was achieved for simulations with this approach.
- **CPU state based on previous CPU state**: The CPU state after reset is set to one state previous to when it was in halt. For example, if the processor was in execute state when it was halt, on

reset it is set to decode and if it was decode, it is set to fetch. **61%** pass rate was seen for this approach in simulations.

- **Instruction re-fetch from memory**: Parts of above approaches along with clearing of **mem_state** is followed here. The CPU state after watchdog reset is always **fetch** and all temporary ALU registers like **alu_out**, **reg_op1**, **reg_op2** are also cleared. Unlike previous approaches, here the instruction is made to re-fetch from the memory instead of using the previous values in memory read control logic. A pass percentage of around **97.7** is achieved with this. The remaining simulations still fail because the HALT happened after the instruction already wrote the result to the general purpose register. So, when the instruction is restarted, it uses these corrupted register and fail. Simply not restarting such instructions should theoretically work, but was not implemented.

The simulations discussed above were the **4300**, that were failing with processor abrupt halt failure signature. When the previously discussed mini regression of 10k simulations, which is a representative of the total fault injection campaign simulations were run, a pass percentage of **93.8** was obtained as opposed to **85.5** without the watchdog. It should be noted that, recovering the processor from Halt after fault injection poses a risk in security perspective. The watchdog timer was implemented only as a proof of concept.

Updating Python Simulation Script

Dhrystone was the application program that was used for fault injection campaign. Dhrystone has functions to check for performance of program based on number of clock cycles consumed using processors performance registers. When watchdog is used to reset the processor, the clock count changes and simulations fail due to this mismatch even though there were no other failures. This check has been masked by writing the expected value to the function. All the errors due to checks between processor when its in halt and reference model are also cleared. The state of reference model is set to be same as the processor after it is reset by watchdog. All of this was updated in **python_wrapper.py**, the python simulation script.

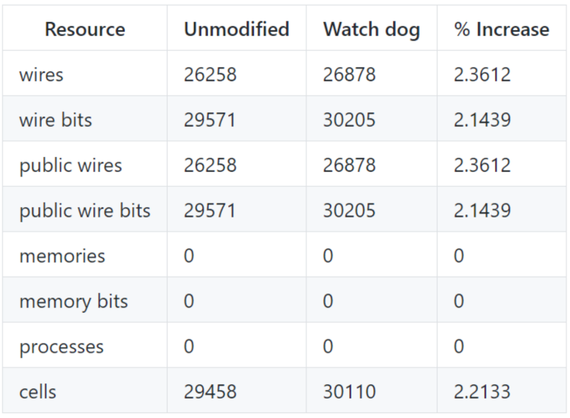
Implementation internals of watchdog

The following registers are used in watchdog internally to implement its functionality,

- **watchdog_counter**: Counts the number of clock cycles the processor has been in halt state. When the processor starts executing again, the count is reset. When the count reaches **78**, the watchdog is activated.
- **watchdog_pc_old**: Holds the address of instruction which was executed before the current instruction. This register is updated on every clock cycle the processor is not in halt and when an instruction completes execution.
- **watchdog_pc_new**: Current Program counter value is saved in this register. Used by watchdog to re-fetch the instruction in-case the processor halts. The PC can also be corrupted by fault injection, so this extra copy is maintained. This register is updated when ever the PC changes.
- **watchdog_set_counter**: Maintains a count indicating how many times the watchdog has restarted the processor.
- **watchdog_sticky**: Used internally by watchdog state machine.

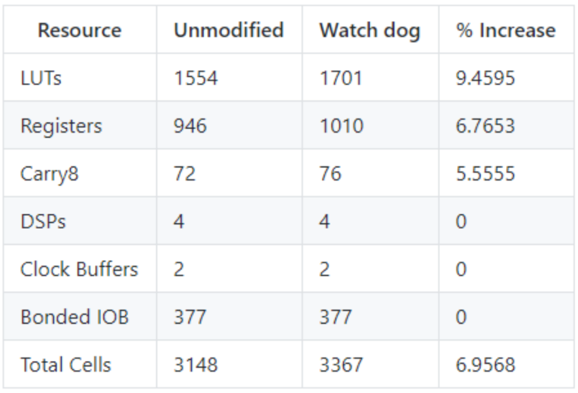
Impact of watchdog on Silicon Area

In order to measure area utilisation of watchdog, the design was synthesised with both **Qflow** and **Vivado**. In **Qflow**, technology used was **osu035_redm4**. Without the watchdog, the maximum frequency achieved in MHz was **96.3177**, with watchdog, it was **96.1666**. The increase in utilisations for various resources was shown in table 5.7a.



Resource	Unmodified	Watch dog	% Increase
wires	26258	26878	2.3612
wire bits	29571	30205	2.1439
public wires	26258	26878	2.3612
public wire bits	29571	30205	2.1439
memories	0	0	0
memory bits	0	0	0
processes	0	0	0
cells	29458	30110	2.2133

(a) Resource usage for Qflow



Resource	Unmodified	Watch dog	% Increase
LUTs	1554	1701	9.4595
Registers	946	1010	6.7653
Carry8	72	76	5.5555
DSPs	4	4	0
Clock Buffers	2	2	0
Bonded IOB	377	377	0
Total Cells	3148	3367	6.9568

(b) Resource usage for Vivado

Figure 5.7: Area usage for Watchdog

With **Vivado**, the maximum operable frequency fell from **424 to 348 MHz**. The target device was set to Zynq-Ultrascale+ and the part was selected to be xczu7ev-ffvc1156-2-e. As can be seen in table 5.7b, the utilisation of total cells has increased by around **7%**. A sample code of the implemented watchdog timer is presented in Appendix C.

5.5. Discussion

Fault injection into two RISC-V processors was performed, namely PicoRV32 and DarkRISCV. The design instrumentation framework was used to instrument both the designs without any porting required. Fault simulation and automatic classification framework on the other hand required adaption based on microarchitecture of the processor as expected. Vulnerability analysis involves root causing the logic in the processor that causes a particular fault signature when injected with fault. This analysis was done for all the major fails for both the processors. One of the major bottleneck in fault space exploration is speed and to counteract this multiprocessor support for simulation script was implemented. The two processors were not designed to be tolerant to fault injection. For the top two failure failure signatures in PicoRV32, Abrupt Halt and Branch Not Taken, a watchdog timer as a hardware countermeasure and inverse operation check were implemented. With the countermeasure implemented, a decrease of **8.3%** in total simulation failures was achieved.

6 Conclusion

6.1. Summary

Chapter 1 introduces the motivation for simulation based fault injection of microprocessors. The chapter discusses various fault injection techniques currently used by attackers to compromise the security of the device. It introduces various state of the art fault injection frameworks being used for instrumenting the design to enable simulation based fault injection. This thesis mainly focuses on development of frameworks for design instrumentation, fault injection and automatic classification of faults.

Chapter 2 of this thesis gives a brief overview of various types of mechanisms that can induce faults in CMOS devices including defects, ageing and external disturbances. A brief background of the fault categories based on their temporal nature and mechanisms involved is reviewed. This is followed by listing of classes in instruction set architecture faults. Furthermore different categories of fault injection techniques were enumerated and contrasted with each other. Subsequently, fault model types that are modelled at signal level and HDL syntactical structure were given. The chapter finishes with a delve into state of the art frameworks proposed in literature.

Chapter 3 of this thesis introduces the reader to instruction set architecture internals of RISC-V. RISC-V is an emerging open source ISA, that is being increasingly being adopted by the industry as well as academia. Thereafter various RISC-V tool chains available along with their current support of variants of RISC-V ISA was examined. This is followed up with an overview of two RISC-V open-source cores, PicoRV32 and DarkRISCV. Their implementation details along with their architecture is illustrated.

Chapter 4 presents the methodology chosen for fault instrumentation and simulation. The chapter initially presents an overview of the Verilog design instrumentation framework. This is followed up with a discussion on the implemented custom parser and algorithmic description of its design. Furthermore, fault instrumentation of different syntactical structures in Verilog is presented and illustrated with examples. Thereafter, fault models and fault classification used by the fault simulation framework is explained. The chapter then highlights the internal mechanisms of fault simulation, log parsing and report generation. Following this, code profiling and optimisations to improve the performance of fault simulation framework is discussed. Finally the chapter ends with an explanation on multiprocessor support provided, which can be used to speedup the fault injection campaign by orders of magnitude.

Chapter 5 of this thesis presents the platform setup for the proposed frameworks. Then it discusses PicoRV32's chosen design parameters, processor state variables and fault bucketing. This is followed up with presentation of fault injection campaign results for PicoRV32. Then, the simulation framework code adoption required for DarkRISCV and its design parameters along with fault bucketing and processor state were explained. Subsequently, fault injection results for DarkRISCV were illustrated. Thereafter, a comparison of different fault signatures for both the processors along with their vulnerability analysis is detailed. Implementation of software countermeasure for branch not taken, the complementary double check was presented. Finally, watchdog timer as a hardware countermeasure for abrupt halt was examined. Its improvement in simulation failure rate along with the impact on silicon area was provided.

6.2. Future Work

In this section, recommendations for future work to further improve the topics addressed in the thesis are highlighted.

- **Exploring different processor implementations**: This thesis showed the fault injection of only PicoRV32 and DarkRISCV processors. Several other cores such as E2 from SiFive, RISCY and MRISCV from MIT are available. Their fault injection campaigns may uncover various new classes of fault signatures and give new insights into RISC-V architecture vulnerabilities.
- **Implementation of new fault models**: Only the bit flip fault model was used in fault injection campaigns of the processors. A **XOR** gate was used for this purpose. Replacing the XOR with an **OR**, **AND** gates can yield new fault models of **stuck-at-1** and **stuck-at-0** respectively. Furthermore, a **MUX** can instead be used and choice of selecting the fault model at run-time during the simulation can be provided to the user.
- **Multi-file design support**: Currently the framework can automatically only instrument the designs if they are all present in a single file. Most of the modern designs are modular and are also developed with reusability in mind, making multi file designs very common. The current workaround is to **cat** all the design files into a single file and perform design instrumentation. Support for Multi-file design will further improve the usability of the framework.
- **Support for other design languages**: Only Verilog based designs are supported by the framework. Increasingly designs are being developed in languages such as System Verilog and Chisel. Both of them can be converted to Verilog using tools SV2V and FIRRTL hardware compiler frameworks. Integrating these into the high level flow will future proof the simulation framework.
- **Eliminate Duplicate Injection Signals**: Multiple duplicate signals are possible in injection bus with current parser design. A methodology to identify and removing them should be developed. This will speedup the fault injection campaign by removing simulations which have the same fault.
- **Improved Watchdog**: Watchdog was built as a proof of concept instead of an actual implementation. Hence it was designed to be very specific to PicoRV32 in this thesis. Providing a configurable open source design supporting pipelined processor architectures will help designers to adopt this countermeasure and integrate in their designs.

A
Platform setup

To instrument the Verilog design, initially Pyverilog has to be setup. Pyverilog requires Icarus Verilog, Jinja 2, PLY to be already installed to work. Optionally pytest, pytest-pythonpath, graphviz, pygraphviz can be installed for automatic testing as well as graphical visualisation of data flow and control flow graphs. They can be installed on a linux system by the executing the following on the command line,

```
sudo apt install iverilog              #Install icarus verilog 10.1 or later
pip3 install jinja2 ply                #Install atleast Jinja2 2.10 or PLY 3.4
pip3 install pytest pytest-pythonpath
sudo apt install graphviz
pip3 install pygraphviz
```

Pyverilog can be downloaded from Github repository [2]. Once the requirements are met, install pyverilog by,

```
python3 setup.py install
```

Copy the **custom parser** and **Verilog design** files to example directory in Pyverilog. Run the custom parser by the following,

```
python3 custom_parser.py design.v
```

Copy the generated instrumented design file, **design_modified.v** as well as **mapping.txt** file to fault injection directory, which has the python simulation script **python_wrapper.py**. If not already available, install and configure RISC-V tool chain for **rv32im** architecture by the following,

```
git clone https://github.com/riscv/riscv-gnu-toolchain riscv-gnu-toolchain-rv32i
../configure --with-arch=rv32im --prefix=/opt/riscv32im
make -j4
```

To generate the binary **main_app.data** for **dhrystone** benchmark using the RISC-V tool chain a **Makefile** is provided. Run the Makefile with **dhry** target,

```
make dhry
```

In **fault_inj_sig_list.txt** file, enter names of signals that are to be injected with faults. The format is, a single signal name should be specified per line, without any delimiters, such as

```
decoder_trigger
launch_next_instr
mem_do_prefetch
```

Set the following parameters in python simulation script (python_wrapper.py) according to the chosen design and simulation requirements,

```
inj_period        = 2 #length of the fault, 1 clock cycle
run_time_prog     = 99996 # Simulation runtime in clock cycles x2
signal_list_file = "fault_inj_sig_list.txt" #signals to be injected with fault
mapping_file_name= "mapping.txt" #for inj_bus indexs for design
write_to_csv      = 0 #simulation logs are written in csv format
golden_run        = 0 #set for a golden reference run
regression_run    = 1
gen_vcd           = 0
print_state_to_text_file = 0 #Print Processor trace for every clock cycle
```

The simulation run-time is set to be **99996** as it covers all unique targeted instructions in dhrystone, which were described in section 4.4. **Verilator** is required to convert the instrumented design to a C++ object. Install it by,

```
sudo apt install verilator
```

Simulation and result parsing is controlled by script, named **top**. It can be run by **./top**. It calls the following scripts internally,

```
source python_command #Runs Python simulation script
python3 result_python.py #Script to parse generated simulation logs
```

The **python_command** script runs verilator and python_wrapper.py scripts to generate simulation logs. The simulation logs are generated in the current directory, with each logs size around 4KB if only final run summary is printed and 500KB if processor state for every clock cycle is printed. Log names are unique as they have timestamp of run time, fault injection length, time instance and instruction targeted for FI.The sample names of logs are as shown,

```
1597911810.978404_179_99996_84728_2_3560_3553_cpu_state_inj_in_0x20_srl.txt
1597911812.789443_179_99996_30_2_3560_3553_cpu_state_inj_in_0x02_addi.txt
1597911814.309057_179_99996_36640_2_3552_3545_cpu_state_inj_out_0x10_slli.txt
1597911816.2411432_179_99996_82012_2_3560_3553_cpu_state_inj_in_0x20_remu.txt
1597911818.047203_179_99996_36498_2_3552_3545_cpu_state_inj_out_0x20_jalr.txt
```

For a multiprocessor system, the number of processes run in parallel can be controlled by setting the number of available semaphore to assailable processor cores in **python_wrapper** script. In the following example, 2 processes can be run in parallel,

```
sema = multiprocessing.Semaphore(2)
```

Every simulation takes around **3.56 seconds** to finish. Thus, for a fault campaign with **100k** simulations will take approximately **100 hours** on a system with single core. The final run summary listing total tests run, passed, failed and various fault signatures is generated in **results.txt** by **result_python.py** script, which can be used as a starting point for all analysis.

B

RV32I ISA

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

C

Watchdog Timer - Sample Verilog Design

```verilog
always @(posedge clk_inj) begin
    watchdog_counter <= resetn_inj ? (watchdog_counter + 1) : 0 ;
      if (watchdog_counter == 77) begin
        mem_state <= 0;
      end
      if (watchdog_counter == 78) begin
        reg_next_pc <= watchdog_pc_new;
        cpu_state <= 8'b01000000; // Fetch state
      end
      if (watchdog_counter == 79) begin
        watchdog_sticky <= 1;
      end
      if (watchdog_counter > 79) begin //Restarting only after 80 clocks
        watchdog_counter <= 0;
        cpu_state <= 8'b01000000; // Fetch state

        alu_out = 0; //Clearing ALU registers
        alu_out_q <= 0;
        reg_op1 <= 0;
        reg_op2 <= 0;
        decoder_trigger <= 1;

        watchdog_set_counter <= watchdog_set_counter+1;
        watchdog_sticky <= 0;
      end
      if ((watchdog_pc_new != reg_pc) & (watchdog_counter < 80)) begin
        watchdog_pc_new <= reg_pc;
        watchdog_pc_old <= watchdog_pc_new;
        watchdog_counter <= 0;
      end
end
```

Bibliography

[1] DarkRISCV. URL `https://github.com/darklife/darkriscv`.

[2] Pyverilog. URL `https://github.com/PyHDI/Pyverilog`.

[3] RISC-V ISA. URL `https://riscv.org//wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf`.

[4] Secure Devices Shipment Trends in Europe. URL `www.eurosmart.com/eurosmarts-secure-elements-market-analysis-and-forecasts/`.

[5] Jaume Abella and Xavier Vera. Electromigration for microarchitects. 2010.

[6] Israel Koren et al. Alessandro Barenghi, Luca Breveglieri. Countermeasures against fault attacks on software implemented aes: Effectiveness and cost. 2010.

[7] Luca Breveglieri et al. Alessandro Barenghi, Guido M. Bertoni. Low voltage fault attacks to aes. 2010.

[8] Luca Breveglieri et al. Alessandro Barenghi, Guido M.Bertoni. A fault induction technique based on voltage underfeeding with application to attacks against aes and rsa. 2013.

[9] Giorgio Di Natale Paolo Prinetto Alfredo Benso, Stefano Di Carlo. A watchdog processor to detect data and control flow errors. 2003.

[10] Todd Austin Andrea Pellegrini, Valeria Bertacco. Fault-based attack of rsa authentication. 2010.

[11] Marc Witteman Bilgiday Yuce, Patrick Schaumont. Fault attacks on secure embedded software: Threats, design, and evaluation. 2018.

[12] OVE S. Taune et al. C. Fibich, P. Rössler. A netlist-level fault-injection tool for fpgas. 2015.

[13] Jaco Hofmann et al. Carsten Heinz, Yannick Lavan. A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors. 2019.

[14] Hyungmin Cho. Impact of microarchitectural differences of risc-v processor cores on soft error effects. 2018.

[15] Hugues Thiebeauld Christophe Giraud. A survey on fault attacks. 2004.

[16] A. DeMillo D. Boneh, R and R. J. Lipton. On the importance of checking cryptographic protocols for faults. 1997.

[17] J.-Carlos Baraza-Calvo et al. Daniel Gil-Tomás, Joaquín Gracia-Morán. Injecting intermittent faults for the dependability assessment of a fault-tolerant microcomputer system. 2016.

[18] Gerd Ascheid et al. David Kammler, Junqing Guan. A fast and flexible platform for fault injection and evaluation in verilog-based simulations. 2009.

[19] Kwangki Ryoo Dennis Agyemanh Nana Gookyi. Selecting a synthesizable risc-v processor core for low-cost hardware devices. 2019.

[20] Debdeep Mukhopadhyay et al. Durga Prasad Sahoo, Sikhar Patranabis†. Fault tolerant implementations of delay-based physically unclonable functions on fpga. 2016.

[21] Ingrid Verbauwhede Dusko karaklajic, Jorn Marc. Hardware designer's guide to fault attacks. 2013.

[22] Michael Barrow Dustin Richmond and Ryan Kastner. Everyone's a critic: A tool for exploring risc-v projects. 2018.

[23] P. Calvel et al. E. G. Stassinopoulos, G. J. Brucker. Charge generation by heavy ions in power mosfets, burnout space predictions and dynamic seb sensitivity. 1992.

[24] M. Rimen J. Ohlsson J. Karlsson E. Jenn, J. Arlat. Fault injection into vhdl models: The mefisto tool. 1994.

[25] Zavier Aguila Eric Matthews and Lesley Shannon. Evaluating the performance efficiency of a soft-processor, variable-length, parallel-execution-unit architecture for fpgas using the risc-v isa. 2018.

[26] T. Granlund. Defeating modexp side-channel attacks with data-independent execution traces. 2013.

[27] E San Millán et al. H Martin, T Korak. Fault attacks on strngs: Impact of glitches, temperature, and underpowering on randomness. 2014.

[28] DAVID NACCACHE MICHAEL TUNSTALL HAGAI BAR-EL, HAMID CHOUKRI and CLAIRE WHELAN. The sorcerer's apprentice guide to fault attacks. 2006.

[29] Rafic Ayoubi Haissam Ziade and Raoul Velazco. A survey on fault injection techniques. 2004.

[30] Ronak Salamat Hamed Abbasitabar, Hamid R. Zarandi. Susceptibility analysis of leon3 embedded processor against multiple event transients and upsets. 2012.

[31] Ali Mili et al. Hany H. Ammar, Bojan Cukic. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. 1999.

[32] Ludger Hemme. A differential fault attack against early rounds of des. 2004.

[33] Schmidt J-M Hutter M. The temperature side channel and heating fault attacks. 2013.

[34] Juan-Carlos Ruiz Ilya Tuzov, David de Andrés. Accurate robustness assessment of hdl models through iterative statistical fault injection. 2018.

[35] Jorn-Marc Schmidt Ingrid Verbauwhede, Dusko Karaklajic. The fault attack jungle - a classification model to guide you. 2011.

[36] Rolf Isermann. Fault-diagnosis systems: An introduction from fault detection to fault tolerance. 2006.

[37] D. Gil J. Gracia, J.C. Baraza and P.J. Gil. Comparison and application of different vhdl-based fault injection techniques. 2001.

[38] Chien-Ning Chen Jakub Breier, Dirmanto Jap. Laser profiling for the back-side fault attacks. 2015.

[39] James C. Hoe et al. Jared C. Smolens, Brian T. Gold. Detecting emerging wearout faults. 2007.

[40] Federico Menarini Jasper G. J. van Woudenberg, Marc F. Witteman. Practical optical fault injection on secure microcontrollers. 2011.

[41] S.Zafar J.H.Stathis. The negative bias temperature instability in mos devices: A review. 2005.

[42] Christophe Deleuze et al. Johan Laurent, Vincent Beroulle. Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a risc-v processor. 2019.

[43] Christian Steger et al. Johannes Grinschgl, Armin Krieg. Automatic saboteur placement for emulation-based multi-bit fault injection. 2011.

[44] Sara Blanc Daniel Gil Juan-Carlos Baraza, Joaquín Gracia and Pedro-J. Gil. Enhancement of fault injection techniques based on the modification of vhdl code. 2008.

[45] M. Lisart et al. K. Tobich, P. Maurine. Voltage spikes on the substrate to obtain timing faults. 2013.

[46] David Kanter. Risc-v offers simple, modular isa. 2016.

[47] Karthik Pattabiraman Layali Rashid and Member Sathish Gopalakrishnan. Characterizing the impact of intermittent hardware faults on programs. 2015.

[48] Niek Timmers Lucian Cojocar, Kostas Papagiannopoulos. Instruction duplication: Leaky and not too fault-tolerant! 2017.

[49] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. 1962.

[50] J. Karlsson et al. M. Rimén, J. Ohlsson. Design guidelines of a vhdl-based simulation tool for the validation of fault tolerance. 1993.

[51] Michail Maniatakos ; Maria K. Michael ; Yiorgos Makris. Investigating the limits of avf analysis in the presence of multiple bit errors. 2013.

[52] J.W. McPherson. Reliability challenges for 45nm and beyond. 2006.

[53] Timothy K. Tsai Mei-Chen Hsueh and Ravishankar K.Iyer. Fault injection techniques and tools. 1997.

[54] E. Encrenaz et al. N. Moro, K. Heydemann. Formal verification of a software countermeasure against instruction skip attacks. 2013.

[55] P. P. Shirvani N. Oh and E. J. McCluskey. Error detection by duplicated instruc- tions in super-scalar processors. 2002.

[56] Amine Dehbaoui et al. Nicolas Moro, Karine Heydemann. Experimental evaluation of two software countermeasures against fault attacks. 2014.

[57] Marc Witteman Niek Timmers, Albert Spruyt. Controlling pc on arm using fault injection. 2016.

[58] X. Kauffmann-Tourkestansky et al. P. Berthome, K. Heydemann†. High level model of control flow attacks for smart card functional security. 2012.

[59] Davide Rossi Pasquale Davide Schiavone, Francesco Conti. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. 2017.

[60] X. Montagner et al. Ph. Cazenave, P. Fouillat. Total dose effects on gate controlled lateral pnp bipolar junction transistors. 1998.

[61] E.J. McCluskey P.P. Shirvani, N. Oh. Software-implemented hardware fault tolerance experiments cots in space. 2000.

[62] Francesco Regazzoni Roberta Piscitelli, Shivam Bhasin. Fault attacks, injection techniques and tools for simulation. 2017.

[63] Dominik Ballek et al. Roland Höller, Dominic Haselberger. Open-source risc-v processor ip cores for fpgas – overview and evaluation. 2019.

[64] Peng Huang et al. Rui Zhang, Calvin Deutschbein. End-to-end automated exploit generation for validating the security of processor designs. 2018.

[65] Skorobogatov S. Local heating attacks on flash memory devices. 2009.

[66] P. Maurine S. Ordas, L. Guillaume-Sage. Electromagnetic fault injection: the curse of flip-flops. 2016.

[67] Donato Kava et al. Sahan Bandara, Alan Ehret. Brisc-v: An open-source architecture design space exploration toolbox. 2019.

[68] Jörn-Marc Schmidt and Michael Hutter. Optical and em fault-attacks on crt-based rsa: Concrete results. 2007.

[69] Pradip Bose Schuyler Eldridge, Alper Buyuktosunoglu. Chiffre: A configurable hardware fault injection framework for risc-v systems. 2018.

[70] Jerry M. Soden and Charles F. Hawkins. Electrical properties and detection methods for cmos ic defects. 1989.

[71] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. 2015.

[72] Karim Khalfallah Thomas Roche, Victor Lomné. Combined fault and side-channel attack on protected implementations of aes. 2011.

[73] Mohammad Mortazavi Vahid Khorasani, Bijan Vosoughi Vahdat. Analyzing area penalty of 32-bit fault tolerant alu using bch code. 2011.

[74] Varadan Savulimedu Veeravalli. Fault tolerance for arithmetic and logic unit. 2009.

[75] Amir Moradi et al. Victor Arribas, Felix Wegener. Cryptographic fault diagnosis using verfi. 2020.

[76] Marc Witteman. Secure application programming in the presence of side channel attacks. 2010.