Chiffre: A Configurable Hardware Fault Injection Framework for RISC-V Systems

Schuyler Eldridge
IBM T. J. Watson Research Center
schuyler.eldridge@ibm.com

Alper Buyuktosunoglu
IBM T. J. Watson Research Center
alperb@us.ibm.com

Pradip Bose
IBM T. J. Watson Research Center
pbose@us.ibm.com

ABSTRACT

The application of modern software design approaches to hardware design has increased the speed at which teams can move and the automation they can employ. The adoption of the former, enabled by Hardware Construction Languages (HCLs), and the promise of the latter (e.g., the extensibility of the FIRRTL compiler) enable new approaches to early stage design emulation and evaluation such as power/timing/area estimation, security verification, and improved FPGA emulation. We detail a further automation framework, developed as a performer of the DARPA PERFECT program, that instruments a hardware design with fault injection capabilities. This work-in-progress framework, *Chiffre*, automates the instrumentation of a hardware design with run-time configurable fault injection logic. Through further proposed automation, Chiffre enables arbitrary fault injection experiments providing insight into the resiliency of a given design while leveraging the 1000× speedup of hardware emulation over software emulation. We detail the Chiffre framework and provide a case study of fault injection into its ease of use for injecting faults into a RISC-V system.

CCS CONCEPTS

• **Hardware → Hardware description languages and compilation**; **Test-pattern generation and fault simulation**;

KEYWORDS

Chisel, Firrtl, RISC-V

ACM Reference Format:
Schuyler Eldridge, Alper Buyuktosunoglu, and Pradip Bose. 2018. *Chiffre*: A Configurable Hardware Fault Injection Framework for RISC-V Systems. In *Proceedings of Second Workshop on Computer Architecture with RISC-V (CARRV'18)*. ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

The design of Application Specific Integrated Circuits (ASICs) requires a delicate balance in terms of when decisions are made in the design process. Early architectural decisions often have impacts that are only evident at the last stages of an ASIC toolflow, e.g., power is traditionally measured reliably only in a placed-and-routed design and reliability often requires a taped-out chip.

One mitigation strategy is simply to *move faster* and *automate more*. The ASIC design process has been empirically shown to be amenable to fast-paced *Agile Development* approaches [13]. Orthogonally, early information about the contribution of architectural and design decisions to straightforward quantities like power, performance, and area as well as more amorphous metrics like reliability and security is key to building good ASICs.

One strategy for collecting such information is *Pre-RTL analysis* that operates not on circuits, but on abstract system components, e.g., a cycle accurate simulator like gem5 [4] coupled with a power/timing/area tool like McPAT [15] and an accelerator design space exploration tool like Aladdin [17]. Pre-RTL approaches enable architects to move faster, explore more of the design space, and provide better product team guidance. While concerns about these approaches exist [16, 21], we assert a project management view: *it's hard to be agile when juggling a hierarchy of abstractions spanning simulators, cycle accurate models, and hardware descriptions.*

Alternatively, a design team lives at one abstraction level. Using an algorithmic abstraction, a designer uses a high level synthesis (HLS) flow to convert an algorithmic description of behavior (written in, e.g., SystemC) to a hardware description (e.g., Verilog). Using a generator abstraction, a designer writes a parameterized hardware description in a *hardware construction language* (HCL), e.g., Chisel [3], that elaborates to a hardware description (e.g., Verilog). While a HLS flow and design using HCLs are very different in spirit and implementation, both approaches adopt the standard software design approach of abstraction to enable teams to move faster.

Towards increased automation, others have adopted software approaches gleaned from compilers, e.g., writing transforms that modify a hardware Intermediate Representation (IR) [10]. The FIRRTL language (Flexible Intermediate Representation for RTL) and compiler [14], one such approach, enables this automated optimization and customization of circuits for tasks including:

- **Backend specific optimizations**, e.g., adding technology-specific SRAMs for an ASIC design
- **State snapshotting** via instrumented registers and memories to enable sampled power estimation [12]
- **Faster emulation** via a relaxation of timing parameters and software simulation of certain hardware [11, 12]
- **Timing and area estimation** via static analysis passes [6]
- **Design-time security checks** with a SAT solver verifying a FIRRTL-emitted information flow description [7]

Under the auspices of the DARPA PERFECT program, we have developed low power and resilient design techniques to composably augment a RISC-V microprocessor [8]. One aspect of these resilient design efforts, mentioned but not described in detail in [8], is an open source, FIRRTL-based framework for injecting faults into emulated designs to measure resiliency. This framework, *Chiffre*,[1] then enables *what if* fault injection experiments as well as more

CARRV'18, June 2018, Los Angeles, CA USA
2018.

[1] Chained Hierarchical Injection for Fault Resiliency Evaluation

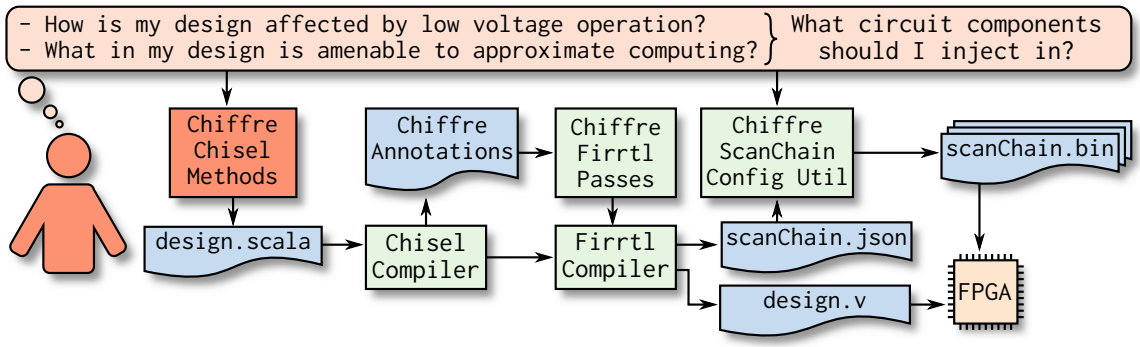

Figure 1: *Chiffre* provides a framework for design-time *instrumentation* and run-time *injection* to answer high-level questions capable of being expressed as fault injection experiments. A designer specifies circuit components in which to inject run-time programmable faults. Chiffre FIRRTL annotations communicate to Chiffre FIRRTL passes what to instrument and how. A designer then configures these injectors at run time with the help of a scan chain bitstream generation utility.

complicated automated analyses analyses, e.g., selective latch hardening based on empirically measured resiliency like in the CLEAR methodology [5]. This paper details the Chiffre framework, provides an example case study using injections into a RISC-V Rocket microprocessor [1], and contrasts this approach with internal approaches to achieve the same goal.

2 MOTIVATION

Figure 1 shows a motivating story where a user wants to answer difficult questions related to a specific hardware design, e.g., how a given design is affected by low voltage operation or how amenable a design is to approximate computing. Certain questions can be reformulated in terms of fault injection experiments, e.g., modeling low voltage operation as faults injected into the N most critical paths. At this point, the user currently has two major options to add fault injection capabilities: manual or automated instrumentation.

Manual instrumentation adds to the cumbersome project management task with yet another model: a new design variant, manually modified with fault injectors that is inherently brittle and must by synced with the original design. Automated instrumentation for hardware is possible [18, 19], but the tools are severely lacking.

Early efforts towards building the Chiffre framework used Verilog-Perl [18], a Perl library with support for reading Verilog netlists into an internal data structure, methods for modifying the data structure, and an emitter to write the data structure back to Verilog. The fundamental difficulty with this approach, similar approaches like Pyverilog [19], or any approach that modifies a high level language, is in the fundamental complexity of the language. Due to the complexity and pitfall-riddled nature of Verilog, SystemVerilog, and VHDL, it is therefore not surprising that Verilog-Perl or Pyverilog do not support all the common synthesizable Verilog constructs. *This is not a criticism of these approaches*—for-profit tool manufactures routinely fail to support modern language features. Similarly unsurprisingly, there exists a cottage industry just for Verilog, SystemVerilog, and VHDL parsing, e.g., the tools provided by Verific Design Automation [2].

Furthermore, our initial Verilog-Perl version operated on FIRRTL-emitted Verilog, i.e., simplistic, structural Verilog that does not stress the limits of the language, and this was still insufficiently powerful. We had to fall back to manually written regular expression parsing and, essentially, macro expansion for insertion of parameterized fault injectors. This resulted in a brittle, non-scalable approach.

The Chiffre framework, a reimplementation of the above efforts in Chisel and FIRRTL, aims to alleviate these difficulties.

3 CHIFFRE FRAMEWORK

Towards enabling fault injection experiments that address difficult high-level questions like those shown in Figure 1, the Chiffre framework provides a number of components that facilitate *design-time instrumentation* and *run-time fault injection*:

- **A Chisel library** that provides example fault injectors as well as traits and methods for annotating the specific circuit components that a user wants to make injectable
- **Chiffre instrumentation FIRRTL passes** that modify a FIRRTL circuit to add run-time programmable fault injectors into annotated components
- **A run-time fault injection utility** for generating scan chain bitstreams that configure available fault injectors
- **An example fault injection controller** implemented as a Rocket Custom Coprocessor (RoCC)

While seamless to use together, the latter two items are entirely independent from the former library—the FIRRTL passes can be used directly on any FIRRTL circuit. Presupposing a converter for a user's hardware design language of choice (e.g., Yosys' Verilog frontend and FIRRTL backend), Chiffre FIRRTL passes can be used to enable fault injection experiments *for any hardware design*.

3.1 Chiffre Chisel Library

The Chiffre Chisel library aims to make fault injection instrumentation as low effort as possible with minimal necessary modifications to a given hardware description. We achieve this aim by *not modifying a circuit at all at the Chisel level*. The Chiffre Chisel library defines two new traits—`ChiffreController` for fault injection controllers and `ChiffreInjectee` for modules that contain fault injectable components. These automatically emit *annotations*,

```
import chisel3._
import chiffre._

/* A controller for injectors on the "main" scan chain */
class MyController extends Module with ChiffreController {
  val io = IO(new Bundle{})
  lazy val scanId = "main"
  /* MyController body with scan chain logic not shown */
}

/* A module with faulty components */
class MyModule extends Module with ChiffreInjectee {
  val io = IO(new Bundle{})
  val x = Reg(UInt(1.W))
  val y = Reg(UInt(4.W))
  val z = Reg(UInt(8.W))
  isFaulty(x, "main", classOf[inject.LfsrInjector32])
  isFaulty(y, "main", classOf[inject.StuckAt])
  isFaulty(z, "main", classOf[inject.CycleInjector32])
}
```

Listing 1: Example source design modified to enable fault injection. Using the Chiffre Chisel library we make registers x, y, and z injectable with, respectively, pseudorandom faults, stuck at faults, and cycle-specific transient faults. These injectors (`LfsrInjector32`, `StuckAt`, and `CycleInjector32`) are added to the `"main"` scan chain controlled by `MyController`.

metadata about a circuit component that a specific FIRRTL pass should operate on, that indicate who is a fault controller and what types of fault injection units to insert.

Listing 1 shows a minimal example of a Chisel circuit where components of `MyModule` are made injectable. The `isFaulty` method annotates these components as faulty, assigns them to a specific scan chain, and identifies a specific fault injector that will be used to inject faults into them. Each scan chain must resolve to an associated scan chain controller (a concrete `Module` that mixes in the `ChiffreController` trait). At this time, no circuit components are connected, no logic is modified, and no fault injectors are inserted.

The Chiffre library additionally provides an abstract base class of a fault injector, `Injector`, and three concrete implementations of fault injectors:

- `LfsrInjector` for pseudorandom bit flips
- `StuckAt` for stuck-at zero/one faults in words
- `CycleInjector` for bit flips at a specific time

Listing 2 shows the annotations resulting from use of the Chiffre library in Listing 1. As should be evident, these are a succinct encapsulation of the information communicated by the modifications. At this point, the FIRRTL compiler takes over and will apply the circuit modifications indicated by these annotations.

3.2 Chiffre Instrumentation FIRRTL Passes

Figure 2 shows the steps the FIRRTL compiler takes to convert a Chisel hardware description to Verilog. The Chisel compiler emits FIRRTL, the FIRRTL compiler optimizes the FIRRTL using passes/transforms, and a Verilog emitter produces Verilog. Transforms may optionally add new annotations for inter-transform

```
[{ "class":"chiffre.passes.ScanChainAnnotation",
   "target":"Top.MyInjector.scan",
   "ctrl":"master",
   "dir":"scan",
   "id":"main" },
 { "class":"chiffre.passes.FaultInjectionAnnotation",
   "target":"Top.MyModule.x",
   "id":"main",
   "injector":"chiffre.inject.LfsrInjector32" },
 { "class":"chiffre.passes.FaultInjectionAnnotation",
   "target":"Top.MyModule.y",
   "id":"main",
   "injector":"chiffre.inject.StuckAt" },
 { "class":"chiffre.passes.FaultInjectionAnnotation",
   "target":"Top.MyModule.z",
   "id":"main",
   "injector":"chiffre.inject.CycleInjector32" }
]
```

Listing 2: Annotations for Chisel code in Listing 1 to be consumed by the `ScanChainTransform` and `FaultInstrumentationTransform`

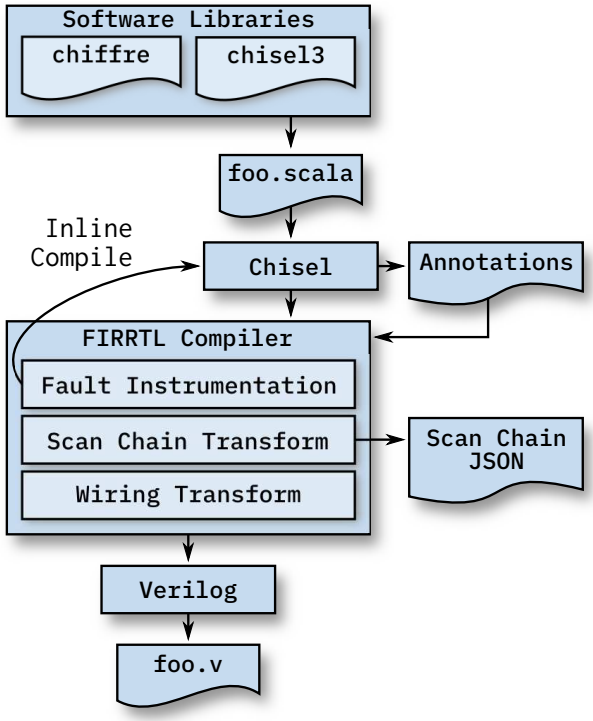

Figure 2: Chisel to Verilog generation process. Chiffre instrumentation occurs using the three transformations shown.

communication. We accomplish Chiffre instrumentation with three specific transforms applied in order:

(1) `FaultInstrumentationTransform`: This splices fault injectors into the circuit and adds them to the scan chain.

```
{ "main":[
  { "name":"Top.MyModule.x",
    "injector":{
      "class":"chiffre.scan.LfsrInjectorInfo",
      "width":1,
      "lfsrWidth":32 }},
  { "name":"Top.MyModule.y",
    "injector":{
      "class":"chiffre.scan.StuckAtInjectorInfo",
      "width":4 }},
  { "name":"Top.MyModule.z",
    "injector":{
      "class":"chiffre.scan.CycleInjectorInfo",
      "width":8,
      "cycleWidth":32 }}
]}
```

Listing 3: JSON description of a Chiffre-generated scan chain for annotations in Listing 2

(2) `ScanChainTransform`: This determines scan chain ordering and determines what should be wired.
(3) `WiringTransform`:[2] This performs all pending connections as determined by previous transforms.

In effect, fault injectors are added, the components that need to be connected and their order is determined, and all pending connections are made. The determined order of the scan chain is exposed externally in a JSON file like that shown in Listing 3.

Figure 3 shows the abstract result of these three transforms. Here, a register `x` was tagged for fault injection just like in Listing 2. The `FaultInstrumentationTransform` adds, internal to the module that contains `x`, an instance of the fault injector specified by the annotation. However, this fault injector may have never existed in the original circuit. As used in MIDAS [11], fault injector modules are added via *inline compilation* of Chisel code. Specifically, the particular injector specified in the annotation is passed through the Chisel compiler and portions of the FIRRTL compiler to generate FIRRTL that is added to the original design. This avoids having to write brittle FIRRTL and takes advantage of the inherent parameterization made available to Chisel by the Scala language.

Once inserted into the circuit, the faulty component (register `x`) is connected to the input of the fault injector. All references to the old component (i.e., any time that `x` appeared in the right-hand-side of a connection statement) are replaced with references to the output of the fault injector. This process repeats for all provided annotations, e.g., in the case of Listing 2 this proceeds for registers `y` and `z`. Finally, all added injector instances emit annotations indicating that they should be added to the scan chain.

The `ScanChainTransform` then collects all scan chain annotations and determines an appropriate ordering for the scan chain. During this process, a JSON description of the scan chain is emitted (see Listing 3) that will be used to generated scan chain bitstreams by downstream tools. Note that the scan chain transform performs no circuit modifications—it only emits further annotations to be consumed by FIRRTL's `WiringTransform`. This transform resolves arbitrary cross module connects described by annotations. In effect, this acts as a primitive on top of which more complex modifications, like the scan chain insertion done here, can be built.

Following this, the FIRRTL compiler completes its usual set of remaining passes and uses its backend emitter to generate Verilog. The original design instrumented with run-time configurable fault injectors is now ready. Note that an unmodified design can be generated by simply not running the Chiffre FIRRTL passes.

3.3 Run-time Fault Injection Utility

For an instrumented design, a user needs to generate and scan in a scan chain bitstream that configures all fault injectors in the design. We provide a utility, `scan-chain-config`, that helps generate this bitstream based on user requirements. This utility reads a scan chain description (e.g., Listing 3) generated by `ScanChainTransform`. This operates on a simple principle: *all fault injectors, as part of their Chisel description, must declare what configurable fields they contain, if any.* The utility, given the scan chain description that specifies what fault injectors are used, can then determine what configurable fields exist in a given design. The utility then takes unspecified *free* fields and *binds* them to hard values based on defaults or user-specified parameters.

As an example, the `LfsrInjector` has two fields parameterized by the width of the LFSR: a `Difficulty` and a `Seed`. When using the tool, the user must specify a probability of a fault being injected (which will be converted to a difficulty) and may optionally specify a seed. If the utility doesn't have enough information to bind all free fields, then it fails noisily.

Listing 4 shows the verbose output when configuring the scan chain from Listing 3. The additional configurable fields are shown with bound values. Additionally, the utility generates a header composed of the scan chain length and a Fletcher 32-bit checksum of the non-header portion of the bitstream.

With this utility, the overall fault injection methodology involves generating a set of bitstreams to appropriately cover the fault injection experiment a user wants to run. Such closed-loop support is not provided as part of the Chiffre framework, we intend to add this support going forward.

3.4 Example RoCC Fault Injection Controller

As one example of a Chiffre controller, we provide `LeChiffre` (*Low Effort* Chained Hierarchical Injector for Fault Resiliency Evaluation). `LeChiffre` is a concrete implementation of a `ChiffreController` that uses the RoCC interface. This provides a simple API for scanning in a fault configuration from memory, verifying its checksum, and enabling/disabling fault injection capabilities. This controller can be used to inject into other designs or, in the example we elaborate on below, directly into Rocket.

4 ROCKET CHIP INJECTION CASE STUDY

Using the Chiffre framework, we instrument Rocket Chip's source to inject faults in non-critical control and status registers (CSRs). We then construct a number of different tests to verify the correct functionality of the instrumented designs with faults.

Using the same method as shown in Listing 1, `isFaulty`, we add a `StuckAt` injector to the `cycle` CSR, an `LfsrInjector` to the

[2] We refactored Adam Izraelevitz's original `WiringTransform` to support our needs and upstreamed this with FIRRTL 1.1.0.

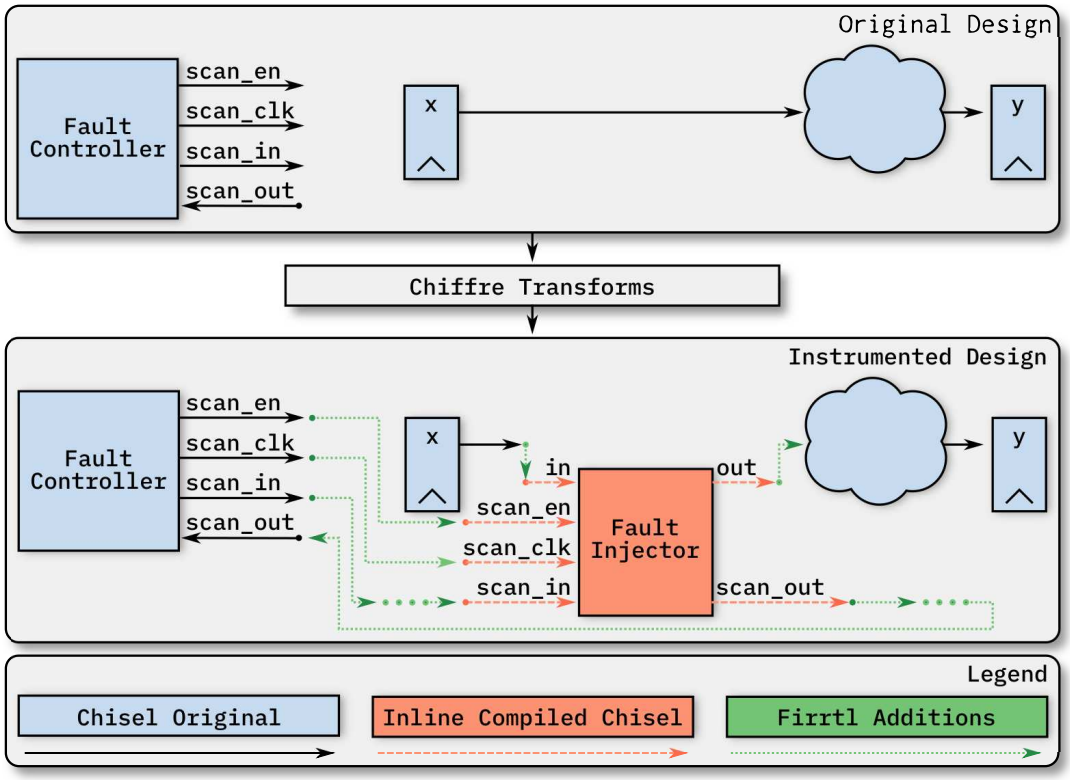

Figure 3: Instrumentation achieved by Chiffre FIRRTL transforms shown in Figure 2. A user-identified component, register x, is instrumented with a fault injector. Inline compilation compiles a specified injector from Chisel to FIRRTL and adds it to the circuit. The fault injector is spliced into the original circuit—x is connected to the fault injector's input and all references to x are replaced with the output of the injector). Finally, all fault injectors and their controller are connected via a scan chain.

`fflags` CSR, and a `CycleInjector` to the `frm` CSR. Critically, this instrumentation takes the same amount of effort as Listing 1, i.e., four lines: one to mixin the injectee trait and one line apiece to annotate the CSRs as faulty with their associated injector.

We then construct one test per fault injector that pass when a condition that would be impossible for the original circuit occurs. Specifically, we read a fixed value from the cycle counter (`0xdeadbeef`), perform back-to-back reads from `frm` and verify that they differ, and read `fflags` multiple times finding differences. This combined test case, while clearly of a "hello world" nature is 70 source lines of code. Critically, *the circuit instrumentation necessary to enable this fault injection only required four lines*. Furthermore, the instrumentation enabled by FIRRTL is entirely reproducible, immune to odd use of language features, etc. This contrasts dramatically with the original Verilog-Perl approach discussed in Section 2.

Summarily, the FIRRTL version of the Chiffre framework enables low effort, non-brittle fault injection experiments that can support arbitrarily complex designs while running at hardware emulation speeds of at least 1000× faster than software emulation.

5 CONCLUSION

The Chiffre framework is intended to provide extremely easy, scalable experimentation with run-time fault injection. As shown in the case study, the amount of user effort required to instrument a design (in terms of modified lines of code) is, at worst, equal to the number of components a user wants to instrument. Further reductions in this are planned with additional Chiffre traits that instrument all components in a region of the design (e.g., in one specific module or in a module and all its submodules) or all components that match a specific type/attribute (e.g., all registers). This approach enables scalability, from a designer effort perspective, for fault injection experiments that encompass entire designs.

Additionally (and critically), source-level modifications needed by the Chiffre framework make no actual modifications to the source design. These only act as helpers for emitting FIRRTL annotations. Equivalently, a user could write FIRRTL annotations directly or with an auxiliary helper utility while avoiding any of the traits and methods provided by the Chiffre Chisel library. This, coupled with the fact that FIRRTL backends are becoming more common for hardware construction languages as well as hardware description languages (e.g., Yosys' FIRRTL backend), enables the use of the Chiffre framework for designs in arbitrary languages that can be converted automatically to FIRRTL.

Chiffre's instrumentation passes bear a strong resemblance to Strober [12] and MIDAS [11]. Both involve splicing Chisel artifacts

```
main:
  length: 0x50 # scan chain binary length in bits
  checksum: f7a1719a # Fletcher checksum (32-bit)
  raw: 0000f6997ffff300 00000802f7a1719a 00000050
  chain:
  - Top.MyModule.x:
    type: lfsr32 # flip `x` with 50% fault probability
    width: 1
      name: Seed
        width: 32
        value: 00000000000000000000000000000001
      name: Difficulty
        width: 32
        value: 01111111111111111111111111111111
  - Top.MyModule.y:
    type: stuckAt # inject 0x03 into all bits of `y`
    width: 4
      name: Mask
        width: 4
        value: 1111
      name: StuckAt
        width: 4
        value: 0011
  - Top.MyModule.z:
    type: cycle32
    width: 8
      name: Cycle # inject 0x2 at cycle 0x8 into `z`
        width: 32
        value: 00000000000000000000000000001000
      name: CycleInject
        width: 8
        value: 00000010
```

Listing 4: Verbose output of `scan-chain-config` showing bound configurable fields for the fault injectors in Listing 3

(MIDAS' *widgets*) into a design, wiring these up with a network,[3] and exposing information about the network to auxiliary tools that orchestrate complex experiments, e.g., state snapshotting. These similarities may motivate the need for *common instrumentation infrastructure* for the set of automated modifications that involve information injection/extraction from a FIRRTL design.

The Chiffre framework currently only supports fault injection into register or wire components. We plan to add support for static and dynamic memory injection. However, dynamic fault injection into memories may require decoupling emulation time from FPGA clock ticks, i.e., FAME-like transformations [20] that push very close to Strober/MIDAS. Additionally, Chiffre requires further automation related to fault injection data collection, e.g., we would like to move the level of abstraction from, "Which register should I inject into?" to "What is the sensitivity of this register to voltage noise?" The latter requires generation of an array of scan chain bitstreams as well as an easy way to collect the results of a fault injection experiment. We view this broadly as an open question.

Chiffre is an actively developed open source hardware/software project available on GitHub [9] under an Apache v2 license.

[3] Strober/MIDAS necessarily provide wider interfaces than Chiffre's one-bit scan chain. We plan to extend Chiffre similarly to support faster fault injector configuration.

ACKNOWLEDGMENTS

We would like to thank Adam Izraelevitz for his implementation of the original `WiringTransform` and Jack Koenig for his helpful review of our refactored version. This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This document is Approved for Public Release, Distribution Unlimited.

REFERENCES

[1] Krste Asanovic, et al. 2016. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).
[2] Verific Design Automation. 2018. Verific Design Automation Website. Online: https://verific.com. (2018).
[3] J. Bachrach, et al. 2012. Chisel: Constructing hardware in a Scala embedded language. In *49th Design Automation Conference (DAC), 2012*. 1212–1221.
[4] Nathan Binkert, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
[5] Eric Cheng, et al. 2016. CLEAR: Cross-layer exploration for architecting resilience: Combining hardware and software techniques to tolerate soft errors in processor cores. In *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.
[6] Intel Corporation. 2017. Rapid Design Methods for Developing Hardware Accelerators. Online: https://github.com/intel/rapid-design-methods-for-developing-hardware-accelerators. (2017).
[7] Shuwen Deng, et al. 2017. SecChisel: Language and Tool for Practical and Scalable Security Verification of Security-Aware Hardware Architectures. Cryptology ePrint Archive, Report 2017/193. (2017). https://eprint.iacr.org/2017/193.
[8] Schuyler Eldridge, et al. 2017. A Low Voltage RISC-V Heterogeneous System. In *1st Workshop on Computer Architecture Research with RISC-V (CARRV '17)*.
[9] IBM. 2017. Chiffre. Online: https://github.com/ibm/chiffre. (2017).
[10] Adam M. Izraelevitz, et al. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*. IEEE, 209–216. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8167715
[11] Donggyu Kim, et al. 2017. Evaluation of RISC-V RTL with FPGA-Accelerated Simulation. In *1st Workshop on Computer Architecture Research with RISC-V (CARRV)*.
[12] Donggyu Kim, et al. 2016. Strober: fast and accurate sample-based energy simulation for arbitrary RTL. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 128–139.
[13] Yunsup Lee, et al. 2016. An agile approach to building risc-v microprocessors. *IEEE Micro* 36, 2 (2016), 8–20.
[14] Patrick S. Li, et al. 2016. *Specification for the FIRRTL Language*. Technical Report UCB/EECS-2016-9. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html
[15] Sheng Li, et al. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*. IEEE, 469–480.
[16] Tony Nowatzki, et al. 2014. gem5, GPGPUSim, McPAT, GPUWattch, "Your favorite simulator here" Considered Harmful. (2014).
[17] Yakun Sophia Shao, et al. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 97–108. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6847316
[18] Wilson Snyder. 2018. Verilog-Perl. Online: http://git.veripool.org/git/Verilog-Perl. (2018).
[19] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing (Lecture Notes in Computer Science)*, Vol. 9040. Springer International Publishing, 451–460.
[20] Zhangxi Tan, et al. 2010. A case for FAME: FPGA architecture model execution. In *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, André Seznec, Uri C. Weiser, and Ronny Ronen (Eds.). ACM, 290–301.
[21] Sam Likun Xi, et al. 2015. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 577–589.