Lab-0x01
Hack the Wallet

Marvin Sass
Security in Telecommunications (SecT)

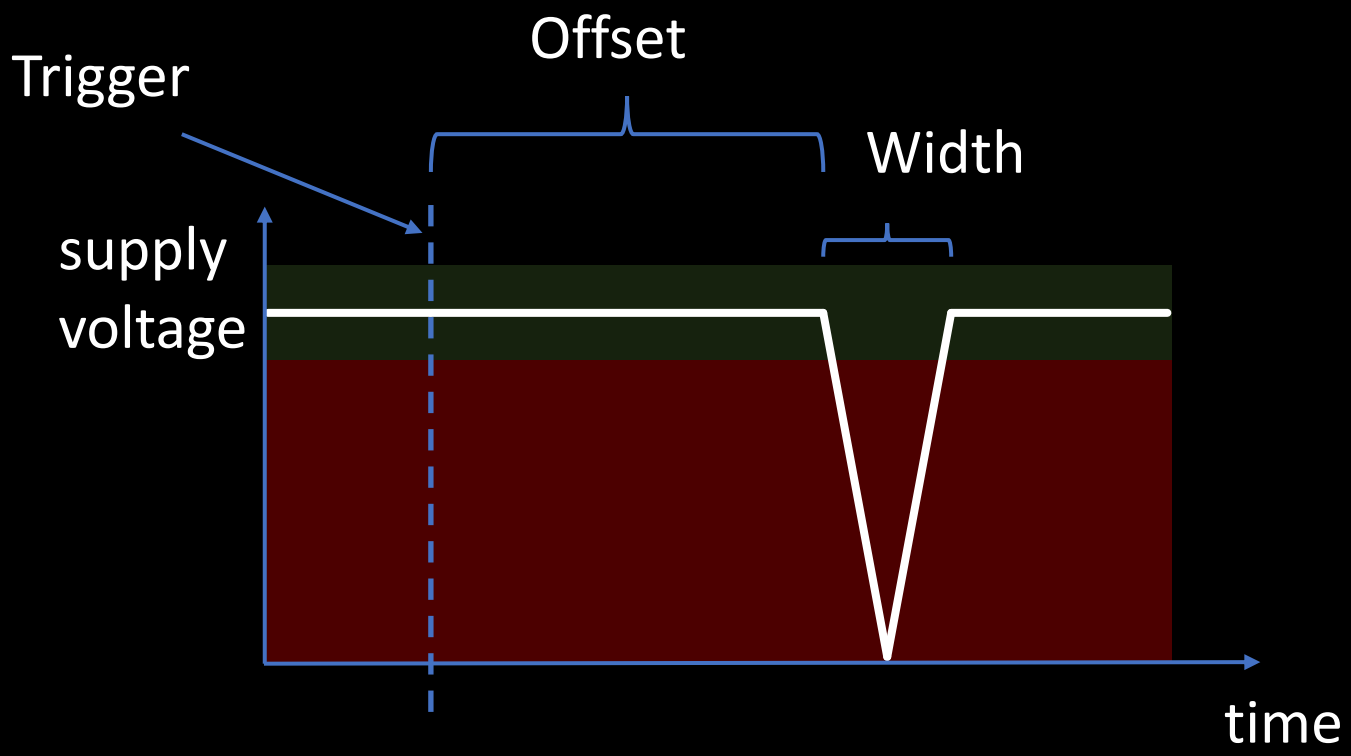


Table of Contents

Before attending the Labs:

You should read this lab sheet to completion and understand every of the tasks you are supposed to work on. If something is not evident to you or you spot some mistake, feel free to reach out to me and I will update this lab-sheet for everybody.

In theory, you may already finish many tasks up front before attending the lab and almost only do the fault injection measurements at our place. This is up to you.

You should also look at the provided resources in parallel (Datasheets, HDL Block designs, Schematics).

Lab-0x01: Overview

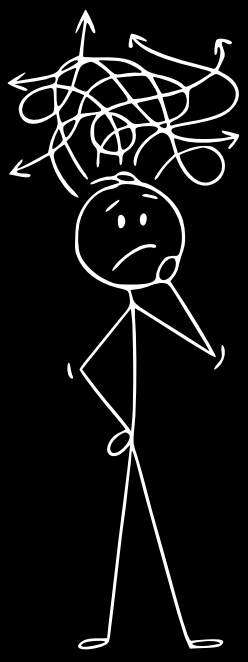
For this second lab, your task is to perform Voltage Fault Injection Attacks against general purpose microcontroller (it's our ATMEGA328P, again 😊) with the ultimate goal of disclosing secret data contained within a read-protected memory region. For this you are going to work in groups of two people, as you have signed up online.

Lab-0x01 will be different from Lab-0x00:

- There are still subtasks, however, they certainly built on top of each other.
- I won't take you by the hand as much as with Lab-0x00. This task is meant to be a hardware CTF.

The structure of this lab is as follows:

1. Understanding the DuT:
    a. Read the specification and run the target
    b. Disassemble the Firmware to find potential targets and secrets
2. Assemble the Glitcher:
    a. Implement a hardware design dedicated to Voltage Fault Injection
    b. Put the Fault Injection circuitry together on a bread board
3. Voltage Fault Injection:
    a. Characterize the DuT's fault response
    b. Break the DuT's authentication scheme and access the secure memory



Required Resources Checklist

In order to take the lab, you by now should have received:

- ✓ Digital Sampling Oscilloscope (DSO)
  - Either Rigol DS1054Z or DS2072A to monitor trigger and observe the faults being injected
- ✓ A bag containing the following parts:
  - Device under Test
  - Either IceSugar- or IceStick-FPGA Board
  - USB-A to USB-B cable
    - to connect your system to the DSO
  - USB-A to USB-C cable
    - to connect your system to the DuT
    - to connect your system to the FPGA (IceSugar only)
  - SMA-to-BNC cable
    - To measure power traces
  - Jumper-Wires
    - For breadboard circuitry
  - At least one Oscilloscope Probe
    - To record the trigger
  - SMA Connector + Cable
    - To inject the Voltage Fault into the DuT
  - Breadboard:
    - To prototype your glitch-circuitry
  - IRLB8721PBF:
    - Our NMOS-of-choice to perform VFI
- ✓ Binary ATMEGA328P firmware (.ELF) with an “all-0x41” password
  - To identify potential spots to inject a voltage Fault
- ✓ Schematics/Datasheets:
  - DuT-PCB
  - FPGA-PCB
  - NMOS Transistor

My Expectations Towards Students

Same as last time.

In addition, as I realized last time people tend to use LLMs quite intensive:
I'd *recommend* you rather not to do so but instead to apply yourself.
As I can't force you not to use it, let's agree that you do not expect any of us to fix your ChatGPT copy-paste issues during lab time.

Student's Options

Same as last time.

In case you are a student who is encountering hard times during this lab, you have the following options:

- ✗ Try again next year (Opt-Out)
- ✓ Try harder this year (Opt-In)

Part of Opting-In is of course also to ask for help whenever you are stuck!
However, if you don't seriously try, I suppose you are going to encounter hard times.

System Software Requirements by Task

This time the system software requirements are not as restrictive as it was the case with Lab-0x00.

Reverse Engineering

If you never ever have analyzed a binary before, don't worry! It is not as hard as you may think. In this case I recommend you checkout Ghidra or Cutter, both fully open source and intuitive tools for reverse engineering binaries of many different architectures (our AVR-architecture as well). Ghidra was published by the NSA (don't worry in this specific case!) whereas Cutter is based on the command line application radare2. Both are available for all common operating systems. These can be installed from:

https://cutter.re/

https://github.com/NationalSecurityAgency/ghidra

If you did some reverse engineering in the past, you can use any disassembler, decompiler or even an emulator which supports AVR architecture. Whatever helps you understanding a binary file is fine 😊. Sky is the limit!

Assembling the Glitcher

We are going to use the IceStudio-IDE, an Open-Source development environment to design, synthesize and implement a hardware design for a small set of commercial FPGAs.

IceStudio is available for all common operating systems and can be downloaded from:

https://icestudio.io/#1k-download

If you have already some HDL experience, you may as well checkout the Yosys workflow:

https://github.com/YosysHQ/yosys

IceStudio itself is built upon the Yosys tooling and gracefully hides the interaction behind a nice and simple graphical user interface.

Attacking the Wallet

For this one you require a recent python installation as well as your favorite code editor. It is further required to install some Python packages on the system.

Voltage Fault Injection Setup

As presented in lecture on Fault Injection Attacks, a simple Voltage Fault Injection Attack setup consists mainly of four different components:

1. Device under Test (DuT)
2. NMOS transistor
3. Configurable pulse generator
4. Controlling lab PC

The functionality each of which are briefly elaborated in the following:

Device under Test (DuT)

The Device under Test (DuT) in this lab will be represented by an ATMEGA328P microcontroller designed by Atmel. It is operating at a clock frequency of 16 MHz (external Oscillator). This time we target a cryptocurrency wallet implementation.

NMOS Transistor

The NMOS transistor is used to inject a parameterized Voltage Fault into the DuT's power supply. The Gate of the NMOS driven by a configurable pulse generator.

Configurable Pulse Generator

In order to control the NMOS switching to inject parameterized voltage perturbations, we need to have a possibility of generating pulses of different widths at different point in time. The configurable pulse generator will be implemented by using any of the available FPGAs.

By controlling the pulse-offset w.r.t. an incoming, time-invariant trigger signal we control at which point in time the voltage fault is being injected.

By controlling the pulse-width, we control the duration of the injected voltage perturbation.

Controlling Lab PC

The controlling lab PC requests the Device under Test to perform certain computations to be attacked and manages the synchronization between fault injection circuitry and the Device under Test. It stores all fault injection results for analysis.

General Fault Injection Procedure

Most commonly, fault injection attacks are performed in two stages:

Stage 1: Fault Characterization

In the first stage, the device under test is executing a test-routine (e.g., an additive loop). By injecting faults characterized by different parameters (offset and width for voltage fault injection), we try to understand how the device under test reacts to the different physical faults being injected. The hereby gained knowledge is utilized in the second stage.

Stage 2: Exploitation

After successful characterization of the DuT's behavior upon fault injection, this knowledge is now materialized to attacks the device. We refer to this as exploitation stage. In case characterization is not viable, you'd have to start exploitation phase with less internals and just "hope for the best". Most of the times it works as well, due to the complexity of fault models. However, the search space certainly increases.

Voltage Fault Injection Setup

No matter what stage you are at, the Fault Injection Setup looks the same. The following figure depicts a typical voltage fault injection setup. The steps to perform a single voltage fault injection attempt are enumerated and will be briefly explained.

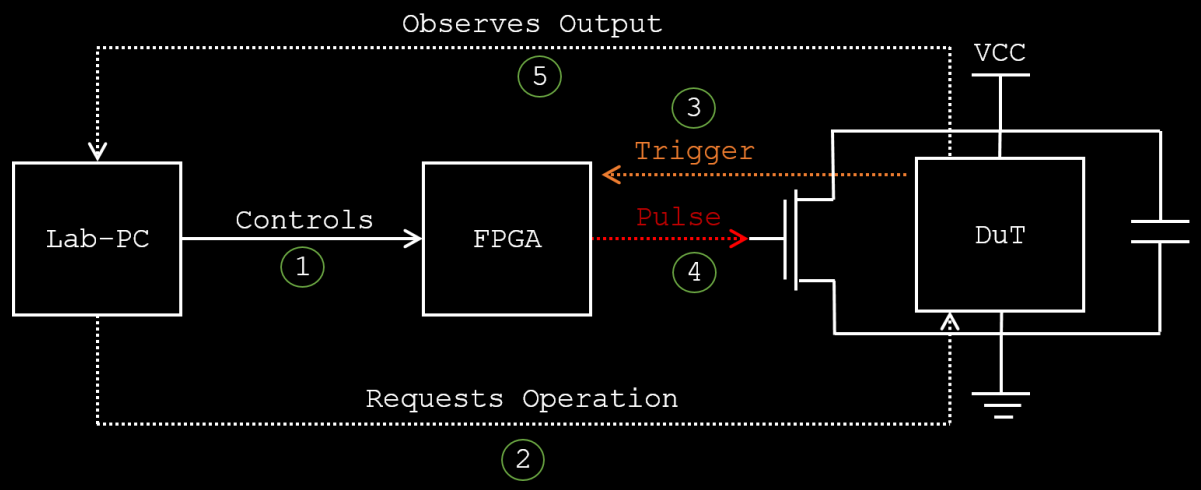


Figure 1: Typical Voltage Fault Injection Setup

Steps to perform a *single* voltage fault injection attempt are:

1. Lab-PC configures the pulse parameters
2. Lab-PC requests DuT to execute code you'd like to attack
3. The FPGA gets triggered by the DuT executing the target routine
4. Based on the configuration, a pulse is fed into the gate of the NMOS
    a. This will short VCC to GND for a controlled period of time!
5. Lab-PC checks the output for any deviations caused by the injected fault and stores the result with the FPGA configuration for later analysis.

Subtask 1:
Understanding the DuT

Specification

The Device under Test represents a wallet for crypto-currencies. The wallet has been implemented on ATMEGA328P. There is a secret password configured by its rightful owner that is stored on the microcontroller's secure memory used for authentication. Further, the Seed-Phrases (24 English words) used to recover the owner's cryptographic wallet-keys are stored within secure memory as well. Such secure memory can only be accessed if the secret password is known.

Moreover, in order to make this wallet really secure, certain security measures have been integrated:

- The passwords have been chosen as random 64 bytes printable ASCII, to prevent dictionary attacks
- Any sort of debug and JTAG port have been fully disabled
- Any unauthorized access to the secure memory results in locking-down of the wallet

However, you were *really* lucky: By social engineering techniques you convinced a regular sales-employee to send you the binary firmware of a wallet-prototype, which has a password of all A's.

Owners of such crypto-wallet may interact with it in several ways:

- Ask about their authorization status (AUTHORIZED, UNAUTHORIZED)
- Request to be authorized by providing a password
- Access their password and seed in secure memory once AUTHORIZED
- Ping the wallet to see if it is up and running

Your goal is to get the rightful owners Seed-Phrases in order to recover the cryptographic keys based on which you are going to transfer all his Bitcoins to your own account (he has Dogecoins as well, but who cares for those?).

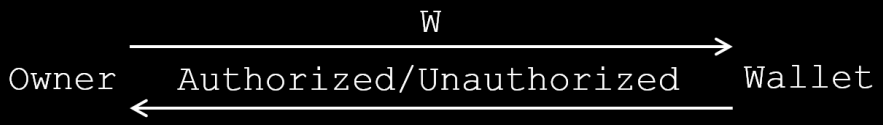
Due to the embedded nature of the wallet, all communication takes place via RS233 (UART).

Communication Overview

As described previously, owners of the wallet may interact with it in several ways. We are going to explain how the intended communication takes place.

Request authorization status

Owners of a wallet may send a 'W' (0x57) to the device in order to request the authorization status. The wallet is going to respond with "Authorized", if the user authorized himself during the current session or by "Unauthorized" if the owner did not yet authorize by providing a valid passphrase.

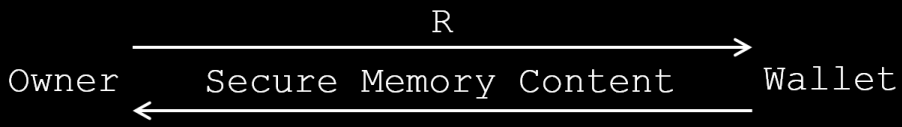


Request authorization

Owners of a wallet may send an 'A' (0x41) to the device in order to initiate an authorization. The command must be followed 64 bytes of password. If the password is correct, requesting the authorization status ('W') will return 'AUTHORIZED, otherwise requesting the authorization status will result in 'UNAUTHORIZED'.

Accessing Secure Memory:

In order to access the secure memory a 'R' (0x52) command is issued. If the user has authorized himself, this will return the content of the secure memory (Password & Seeds). If the user has not authorized himself, this results on the device locking itself down to prevent any sort of brute-forcing and dictionary attack.



Ping the wallet:

The owner may also ping the wallet by sending a 'P' (0x50). If a wallet receives a ping, it will always respond with "Wallet Pong" in order to inform the owner that it an operational wallet.

Task 1.1:
Connect the device to USB and open a serial console (e.g. by using one of screen, minicom, putty, picocom, …) in order to communicate with the device. The UART baudrate of the wallet is 115200. Try all the above commands and understand their behavior. What happens if the device locks itself down for unauthorized access of the Secure Memory? Take a screenshot and append it to your lab report!

Firmware Analysis

Firmware analysis is required, as we are in the domain of implementation attacks. Please check out the lecture on Fault Injection attacks again in case you have no idea what we are referring to. Here you learned, that the same source code may lead to different attack scenarios, depending on what machine instructions and architecture it gets compiled to.

This is why in Fault Injection Attacks we are *really* lucky when we have knowledge about the actual machine instructions being executed. In combination with knowledge of the fault injection method's fault model, it is possible to precisely plan our attack.

As mentioned earlier, in this lab you are lucky and have access to a prototype's firmware. The only difference of the prototype wallet being:

- the password required to access the secure memory
- the Seed required to recover the private cryptographic keys

Everything else is *exactly executed as is*.

Task 1.2:
Recapitulate the Fault Model of Voltage Fault Injection attacks (Lecture). Describe in your own words what a fault model describes in the domain of Fault Injection attacks within your lab report. How does the fault model from voltage fault injection attacks differ from that of Laser Fault Injection and why?

Task 1.3:
Using your RE-tools of choice, try to identify undocumented commands. List any additional insight within your report.

Task 1.4:
Using your RE-tools of choice, try to identify potential points which can be used to synchronize your fault injection attack. Explain within your report.

Subtask 2:
Assembling the Glitcher

Overview:

In this sub-task you are going to work with IceStudio in order to put together the fault injection circuitry (a.k.a. Glitcher). To make it easier, we were preparing Block-Designs in IceStudio, which can be imported by clicking on

Menu > File > Import as Block

and selecting the corresponding block design (.ice).

We provide four such designs in total, which are described before continuing to the tasks:

1. UART_RX.ice defines a block design for a UART receiver
2. UART_TX.ice defines a block design for a UART transmitter
3. CONFIG.ice defines, stores and loads all configurable parameters
4. GLITCH_CTRL.ice controls the fault to be injected by means of its provided inputs

Given this set of verified blocks, your main task here will be to assemble what we refer to as a top module, i.e. a module assembling and interconnecting all the existing hardware blocks.

If you have already some HDL experience or otherwise would like to get there, you can:

- Check out the internals of the Blocks by double clicking on them after importing
- Write your very own Glitcher from scratch for the corresponding FPGA

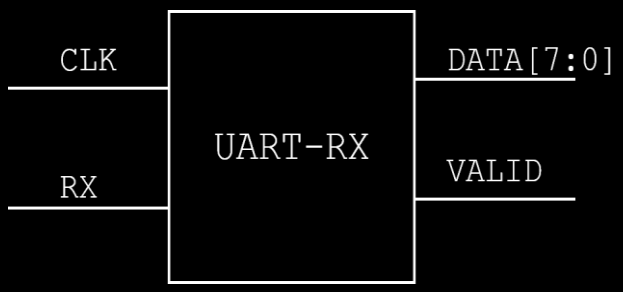
Naturally you'd learn the most by doing it yourself from scratch. However, as we do not have HDL as a preliminary for this course, we provide block designs.

Description of Blocks:

UART_RX.ice

UART_RX.ice defines a hardware design for a UART-Receiver. As both the FPGA boards we are using comprise a USB-to-UART bridge, a UART implementation is required in order to configure the parameters of our Glitcher.

The block-Diagram of the UART-RX module is depicted in the following:

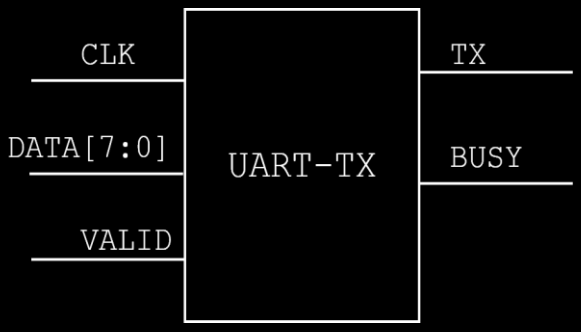


Inputs are shown on the left, outputs are presented to the right.

- CLK represents a port to be connected to the globally synchronizing clock-signal.
- RX represents the output from the corresponding USB-to-UART bridge and an input to the FPGA. Check out the PCB schematic for further information.
- DATA[7:0] represents an 8-bit output port, providing the decoded UART data.
- The VALID port is asserted, whenever an UART transmission has been received, thus indicating whenever a new byte comes in.

UART_TX.ice

In order to read-back parameters from our Glitcher, we require a transmitter module. This is where UART-TX.ice comes into play. The block-Diagram is depicted in the following:



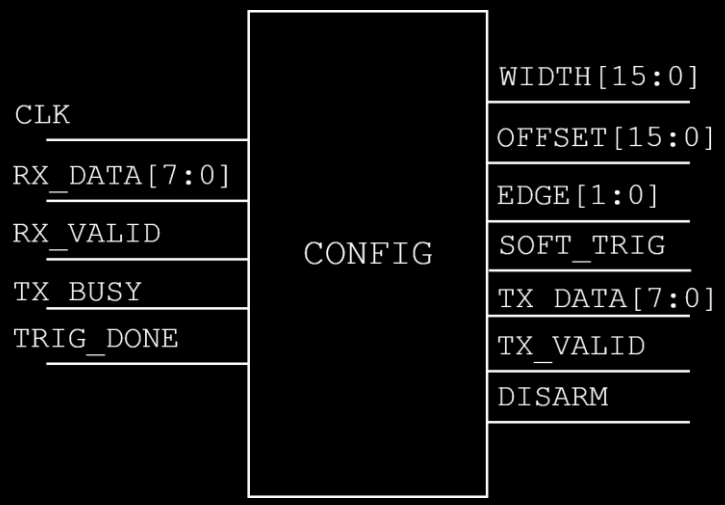
Inputs are shown on the left, outputs are presented to the right.

- CLK represents a port to be connected to the globally synchronizing clock-signal.
- DATA[7:0] in the case of a transmitter represents a data byte to be transmitted by the corresponding USB-to-UART bridge via the TX output pin of the FPGA.

- The VALID port should be driven high for a single cycle, whenever DATA[7:0] should be transmitted.
- TX is the pin transmitting the UART signal. It should hence be connected to an output pin of the FPGA. Check out the corresponding schematic for further information.
- BUSY port indicates, whenever the TX module is busy transferring a byte. VALID should not be asserted, unless BUSY is back to low.

CONFIG.ice

The CONFIG block is what handles the more complex task. This represents the central state machine for configuring the Glitcher's behavior. It must be granted access to UART-RX as well as UART-TX and -based on the UART data- configures the voltage fault parameters accordingly.



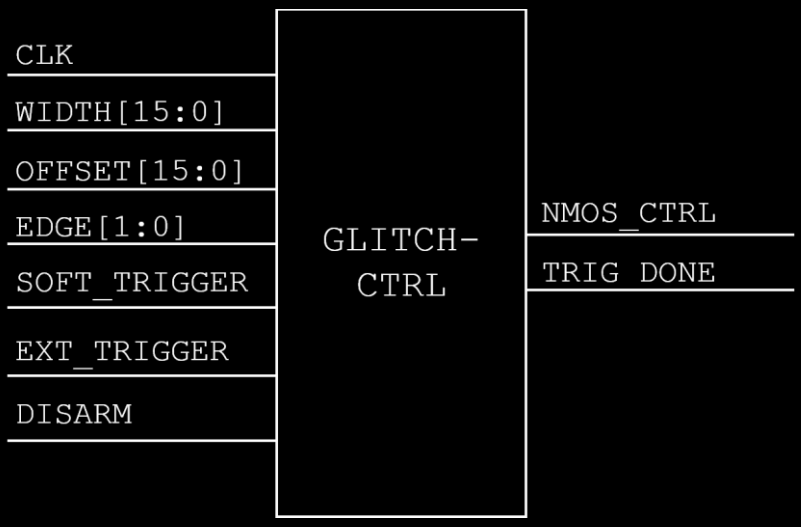
Inputs are shown on the left, outputs are presented to the right.

- CLK represents a port to be connected to the globally synchronizing clock-signal.
- RX_DATA[7:0] represents the byte coming in over UART.
- RX-VALID tells the CONFIG module, that the data at RX_DATA[7:0] port is valid and thus must be sampled.
- TX_BUSY, if asserted high, tells the config module that UART-TX is currently transmitting a byte.
- TRIG_DONE represents a signal to notify the config whenever a TRIGGER has been observed, such that the CONFIG may react in that event.
- WIDTH[15:0] represents the currently configured WIDTH parameter for the voltage fault to be injected.
- OFFSET[15:0] represents the currently configured OFFSET parameter of the voltage fault to be injected.
- EDGE[1:0] provides an encoded bit pattern representative for the trigger edge the FPGA shall react on:
    - Rising edge, falling edge, both edges or software-based trigger
- SOFT_TRIG provides a trigger signal which can be asserted from software by issuing a UART command.

- TX_DATA[7:0] provides data supposed to be written back via the USB-to-UART bridge.
- TX_VALID indicates that the data at TX_DATA[7:0] should be transmitted.
- Finally, DISARM represents an output signal used to disarm the Glitcher based on the selected configuration.

GLITCH_CTRL.ice

The glitch-controller represents the module responsible for injecting and shaping the voltage fault based on the configuration coming from CONFIG. Fault Injection may be initiated by an external trigger signal or by a soft-trigger, i.e. a serial command that internally issues the trigger to be asserted.



- CLK represents a port to be connected to the globally synchronizing clock-signal.
- WIDTH[15:0] configures the voltage fault's width
- OFFSET[15:0] configures the voltage fault's offset
- EDGE[1:0] configures on which trigger to react on:
  - External Trigger: Raising-Edge, Falling-Edge, Both-Edges
  - Soft Trigger: Trigger coming from a serial command
- DISARM: Input signal to disarm the Glitch-Controller. Once disarmed, it will not react on incoming triggers anymore.
- NMOS_CTRL: NMOS Gate Controller. Should be connected to an output pin. Check out the FPGA schematic as well as the NMOS Transistor's datasheet.
- TRIG_DONE: Signal indicating that a trigger event occurred.

API Description

glitch_control.py

gitch_control.py consists of a GlitchController-Class, which handles serial communication with the Glitcher to, e.g., set the voltage faults offset and width properties.

For obvious reasons, glitch_control.py can only be used if you decide to utilize our Glitch-Configuration state machine (CONFIG.ice) state machine and not if you decide to implement your own Glitcher.

Glitch_control.py defines a class referred to as GlitchController, which defines the following functionality:

```
def __init__(self):
    ...
```

Constructor of the GlitchController class. This is going to enumerate all connected USB devices looking for the USB product and vendor ID of the Glitcher. If the Glitcher (FPGA) is not connected, an assertion is generated.

```
def reset(self):
    ...
```

Resets the Glitcher. On reset, all properties are assigned values of zero.

```
@property
def config(self) -> bytes:
    ...
```

This returns the internal config of the Glitcher in bytes. It has the structure `'D' DB DB 'W' WB WB 'M' MB MB`. Where

- DB: Delay Byte
- WB: Width Byte
- MB: Mode Byte

You should not worry too much about this function, as the getter for `pulse_width`, `pulse_offset` and `trigger_on` are already parsing this config for you. 😊 It is listed for completeness.

```
@property
def pulse_width(self) -> int:

    ...
@pulse_width.setter
def pulse_width(self, width: int):
    ...
```

Getter and setter for the voltage faults width property, defined in terms of FPGA clock cycles.

```
@property
def pulse_offset(self) -> int:
    ...
@pulse_offset.setter
def pulse_offset(self, offset: int):
    ...
```

Getter and setter for the voltage faults offset w.r.t. an incoming trigger signal defined in terms of FPGA clock cycles.

```
@property
def trigger_on(self) -> TriggerConfiguration:
    ...
@trigger_on.setter
def trigger_on(self, trigger_config: TriggerConfiguration):
    ...
```

Getter and setter for the Glitcher's Trigger Configuration. Any instance of `TriggerConfiguration` may be selected:

```
class TriggerConfiguration(enum.Enum):
        TRIGGER_INTERNAL                        = 0b00000000
        TRIGGER_INTERNAL_SINGLE_SHOT            = 0b00000100
        TRIGGER_EXTERNAL_RISING_EDGE            = 0b00000001
        TRIGGER_EXTERNAL_RISING_EDGE_SINGLE_SHOT    = 0b00000101
        TRIGGER_EXTERNAL_FALLING_EDGE           = 0b00000010
        TRIGGER_EXTERNAL_FALLING_EDGE_SINGLE_SHOT   = 0b00000110
        TRIGGER_EXTERNAL_ANY_EDGE               = 0b00000011
        TRIGGER_EXTERNAL_ANY_EDGE_SINGLE_SHOT       = 0b00000111
```

TRIGGER_INTERNAL is what we refer to as a "soft-trigger", i.e. when a Trigger-Command comes in, the Soft-Trigger is asserted. It is a soft-trigger in a sense that it is not a hardware-based trigger.

External Triggers (hardware-based triggers) are distinct by the edge to react on. Rising-, falling as well as both edge types can be configured as trigger.

Further, any trigger is differentiated between regular and single shot:

In Single-Shot triggering the Glitcher unarms itself after it has been triggered. In regular triggering mode the Glitcher continuous to react on any further incoming edge detected.

```
@property
def shot_counter(self) -> int:
    ...
```

This one returns an integer-based counter of how many times the trigger has been asserted based on the provided `TriggerConfiguration`. Please check out the configuration cycle below.

```
def trigger_sw(self):
    ...
```

Sends a soft-trigger command. A fault will be injected as long as the TriggerConfiguration is configured to be one of the internal ones.

Configuration Cycle

The configuration cycle defines how the Glitcher can be used. It usually starts with a reset of the Glitcher, which sets all internal values to zero.

As the Glitcher is reset to a TriggerConfiguration of TRIGGER_INTERNAL, it does not react to any outside signals and hence may be configured in that state.

When the Glitcher is in configurable state, pulse_offset and pulse_width may be defined.

Finally, if you would like to synchronize based on an external trigger, the TriggerConfiguration needs be defined. As soon as you move away from TRIGGER_INTERNAL, the Glithcer cannot be reconfigured unless resetting it again (or at least setting the TriggerConfiguration to be TRIGGER_INTERNAL).

Hence, a single and general applicable configuration cycle looks like:

1. reset
2. set pulse width
3. set pulse offset

configure the trigger

Target FPGA

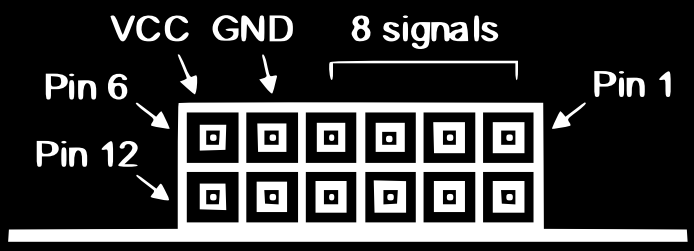
Unless noted otherwise we are going to use IceSugar-nano FPGA by Muse Lab. Schematic as well as Pin Configuration File (.pcf) can be found in the lab resources folder.

This FPGA can be selected in Ice Studio by:

Menu > Select > Boards > LP1K > iCESugar-nano

Pin Description

The Pins in IceStudio are named different from those in the Schematic. As the Glitcher does not require exhaustive I/O functionality, I am only going to give you the Pin-Mapping for the PMOD connector. When you look right at the PMOD connector with the PCB on the bottom, the pins are defined as follows:



Pin Number	IceStudio Naming
Pin 1	PMOD1
Pin 2	PMOD3
Pin 3	PMOD5
Pin 4	PMOD7
Pin 5	No naming, is GND!
Pin 6	No naming, is VCC!
Pin 7	PMOD2
Pin 8	PMOD4
Pin 9	PMOD6
Pin 10	PMOD8
Pin 11	No naming, is GND
Pin 12	No naming, is VCC

Task 2.1: Implement a Glitcher

You should have all the knowledge required to assemble a top module. A top module in the general sense is a module interconnecting all your building blocks.

Load all the .ice blocks in a new project you name TOP.ice and interconnect them accordingly. Define Input and Output pins, with respect to the previous pin definition.

Once you have assembled your top module, use the Python-API we provide in order to implement a quicktest-alike testing of the functionality:

For each configurable parameter of the Glitcher:

1. Create a random, valid value
2. Set the values within a configuration cycle
3. Read back the parameters and check for equality with the written one.

If this test passes, you and your Glitcher are likely ready to continue with the next task. Anyhow, before continuing please consult one of us to check your design.

Task 2.2: Implement Fault Injection Circuitry

Check the datasheet of the NMOS Transistor (IRLB8721PBF)! Which Pin is Source? Which pin in Drain? Which pin is Gate? What about the Exposed pad on the Backside?

Using the provided breadboard, the SMA connector, the NMOS and a couple of jumpers wires, assemble fault injection circuitry which based on an incoming pulse (from Glitcher) shorts VCC to GND.

Let us attempt to counteract Murphey by having some discussion about your circuitry, before continuing! This is important, as you may brick yours, ours or both our hardware if continuing with an erroneous setup.

Subtask 3:
Voltage Fault Injection

After all this initial work you are ready to get started with the real deal! Based on Subtask 1 and Subtask 2 you should have a solid understanding of what is going on. Make sure this is indeed the case!

As in most Fault Injection campaigns, Subtask 3 is split into two parts:

1. Characterization of Fault Behavior
2. Exploitation

Characterization is the process of analyzing the fault behavior for one specific target executing one specific implementation. This takes place by injecting faults of different parameters and analyzing the different outcome.

Exploitation uses the gained knowledge to perform a specific attack.

API Specification

dut.py

dut.py consists of a DuT-Class, which handles serial communication with the wallet. Out of the box it consists of the following function definitions:

```
def __init__(self):
    ...
```

Constructor of the DuT class. It is going to enumerate all connected USB devices looking for the USB product and vendor ID of the wallet. If the wallet is found it is opened, otherwise an assertion violation is generated.

```
def __del__(self):
    ...
```

Destructor of the DuT-class, which makes sure the serial port gets closed properly.

```
def __clear_buffers(self):
    ...
```

Due to faults being injected, invalid data may be generated on the bus which gets buffered. The solution to that is to call this function before any transmission to make sure, that the RS232 buffers are in an expected state.

```
def reset(self):
    ...
```

Due to the faults being injected, the device will certainly crash! This function can be used to reset the device under test as soon as a crash has been detected.

```
@property
def authorized(self) -> bool:
    ...
```

Property to check whether or not the current user is authorized. Exclusively authorized users may access the secure memory.

```
def authorize(self, password: bytes):
    ...
```

Issues an authorization request based on the provided, 64-byte password. If the password is correct a following check on the `authorized` property will indicate True.

```
def read_secure_memory(self):
    ...
```

Reading secure memory and extracting the seeds represents your goal. This is only possible, however, if the user is `authorized` otherwise, calling this function results in a total lockdown of the device under test.

```
def ping(self) -> bool:
    ...
```

Returns True if the device answers 'Wallet Pong', otherwise False.

Task 3.1: Characterization

Based on Task 1.3: Can you think of ways of characterizing voltage fault injection attempt on the wallet?

If so, extend the API (dut.py) by functions to interface with the wallet accordingly. Afterwards perform characterization of your voltage faults by plotting the faults parameters (Offset over Width) and the corresponding outcome (e.g. color coded or 3-Dimensional plot).

From an abstract point of view, possible outcomes applicable to any fault injection scenario are:

1. Injected Fault has no effect at all
2. Injected Fault crashes the Device under Test and it need to be reset
3. Injected Fault leads to a deviation of the expected outcome

If you see absolutely no way of characterizing faults injected into the Wallet, continue with the next task.

Task 3.2: Stealing Crypto

Based on your initial research on the Device under Test and its Firmware you should have found ways of synchronizing your Glitcher.

Task 3.2.1: Plan

Plan on how to attack the Device under Test. Use the figure in Voltage Fault Injection Setup for orientation. Discuss your plan with one of us before continuing.

Describe your attack plan within your report.

Task 3.2.2: Follow Through

Follow through with your plan. In case you succeed, note down the Seed used to recover the crypto wallet and append it to your report. Take notes on encountered issues as well.