Fagnani Pietro
s303490@studenti.polito.it

D'Elia Luca
s296480@studenti.polito.it

Massetti Marco
s301163@studenti.polito.it

Serra Jacopo
s303510@studenti.polito.it

Emiliano Massimo
s301114@studenti.polito.it

Sineo Gioele
s301398@studenti.polito.it

# OPERATING SYSTEMS - Advanced Project

Year 2023

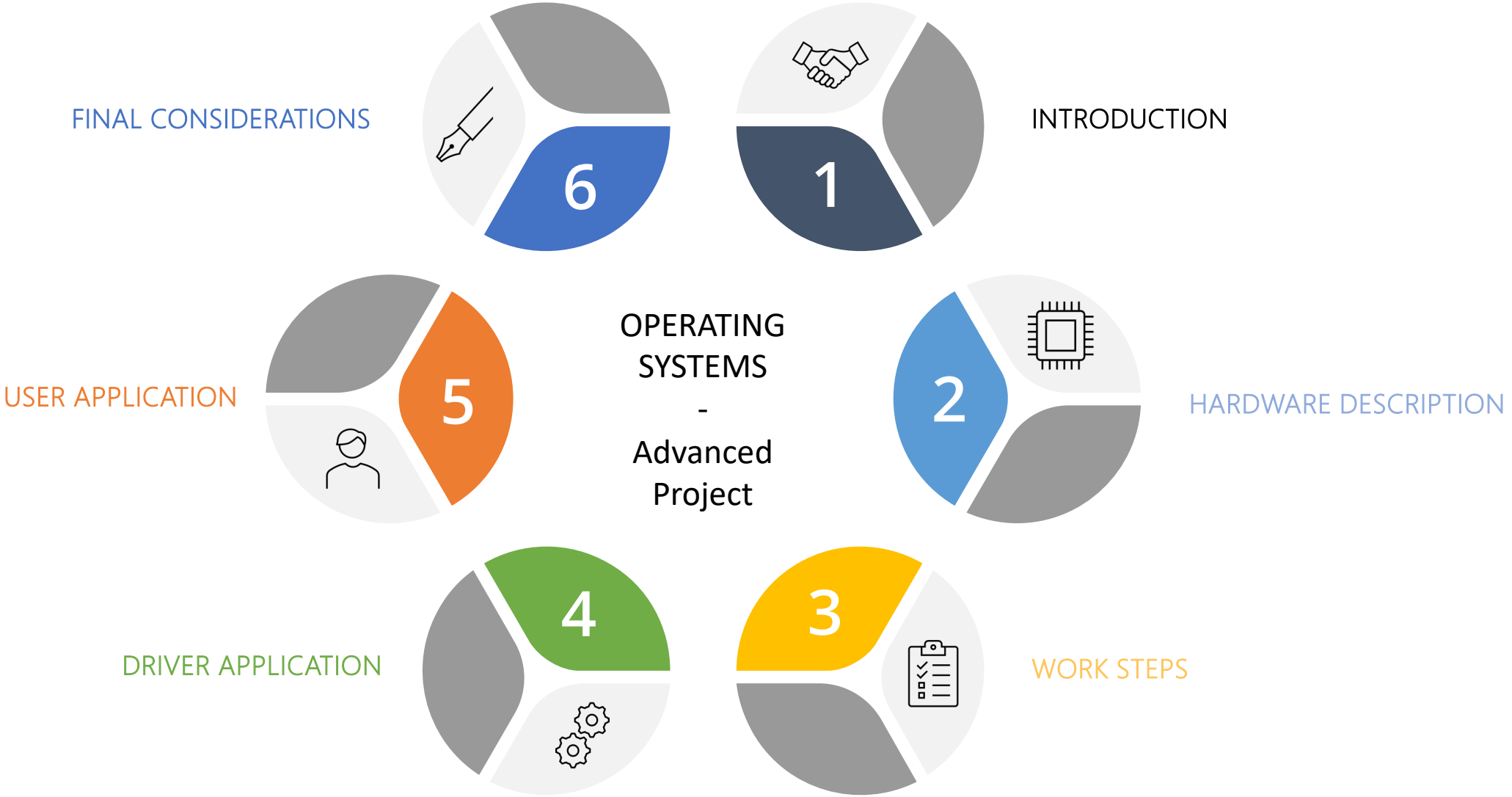
-

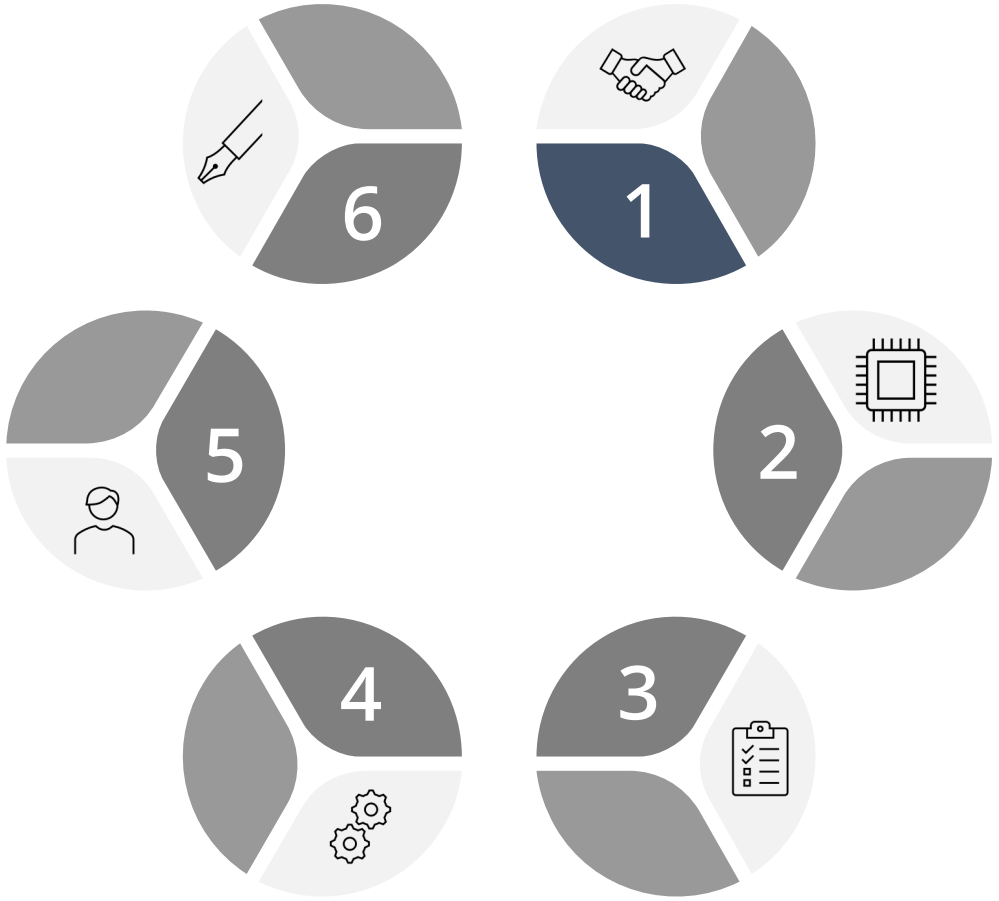
Di Carlo Stefano

Carpegna Alessio

Co-funded by the European Union

Erasmus+


OPERATING SYSTEMS - Advanced Project
1
INTRODUCTION
2
HARDWARE DESCRIPTION
3
WORK STEPS
4
DRIVER APPLICATION
5
USER APPLICATION
6
FINAL CONSIDERATIONS




# 1- Introduction

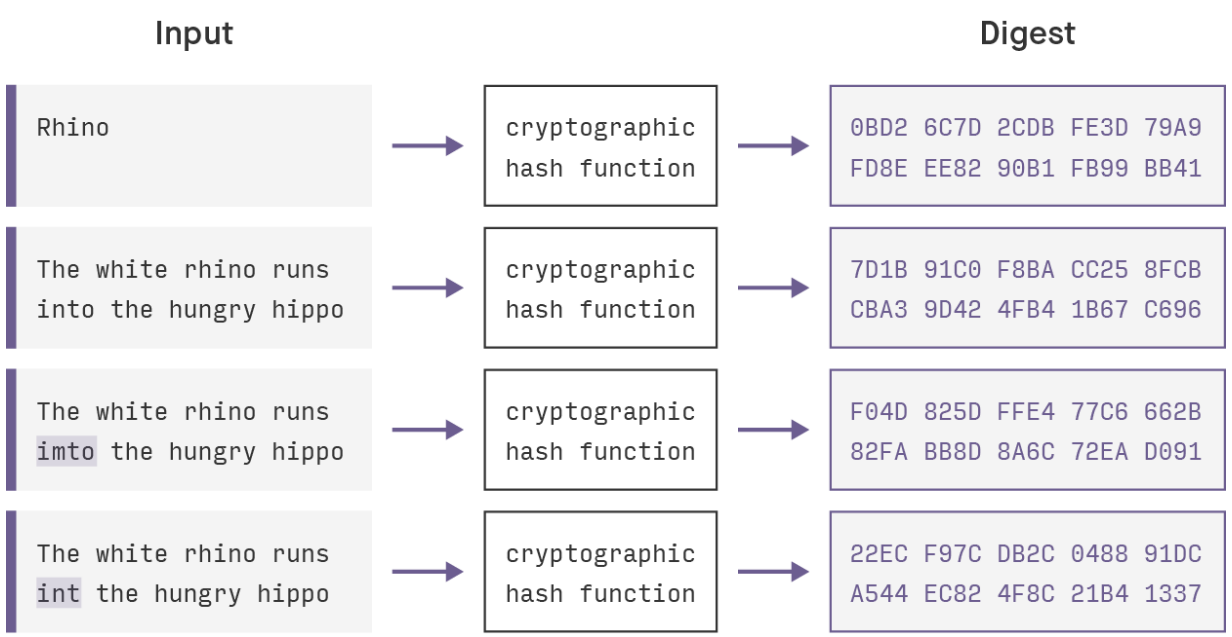
**Development of a Linux device driver for a crypto core in a PYNQ-Xilinx board.**

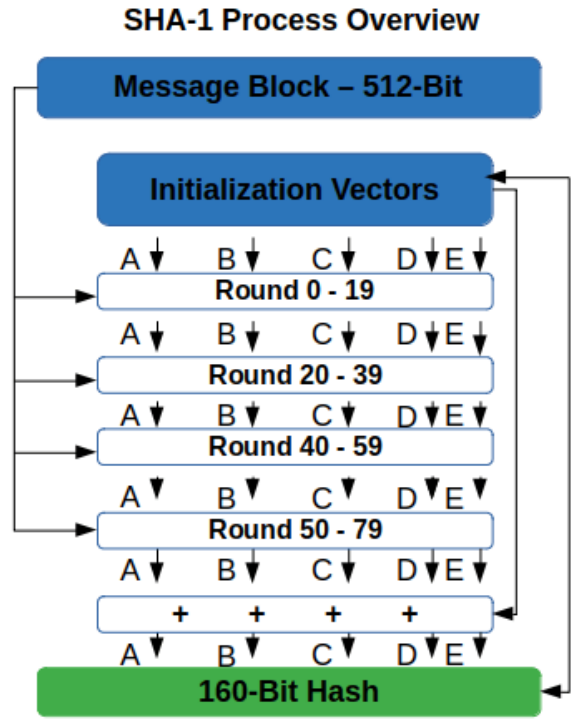
For the development of the driver different steps are required starting from the creation of the hardware accelerator, then to the development of the driver and the user application up to some examples of execution that test the functionality of the entire project.

1

# 1- Introduction

- SHA-1 (Secure Hash Algorithm 1) is a **cryptographic hash function** that processes input data in blocks, applying mathematical operations and transformations to create a fixed-length 160-bit (20-byte) hash value.
  This hash value is a **unique representation** of the input data and is used for data verification and integrity purposes.



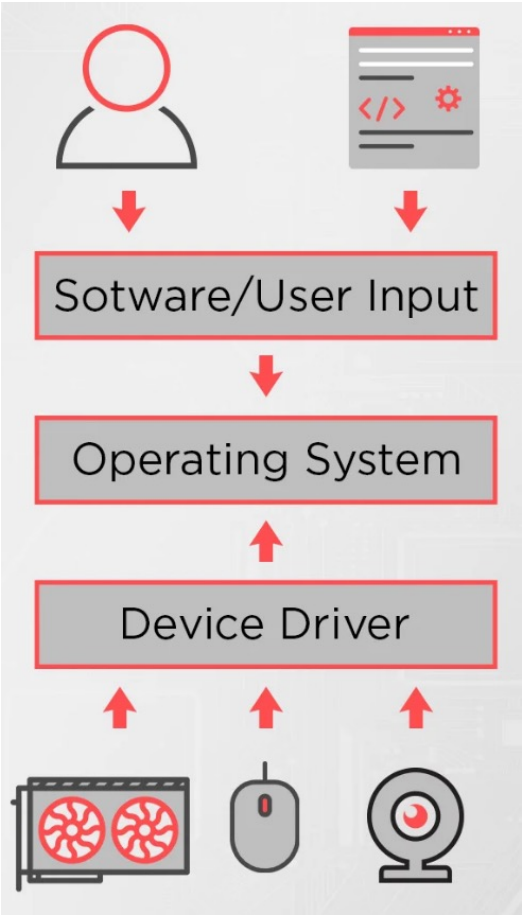



1

# 1- Introduction

- **Driver applications** act as intermediaries between an operating system and hardware components, ensuring effective communication and enabling optimal hardware utilization for users.

- The instructions requested by the user through the user application are **translated into hardware operations** that perform specific functions. This is a necessary job to make custom input/output communicate with the processor or micro-controller used at that time.



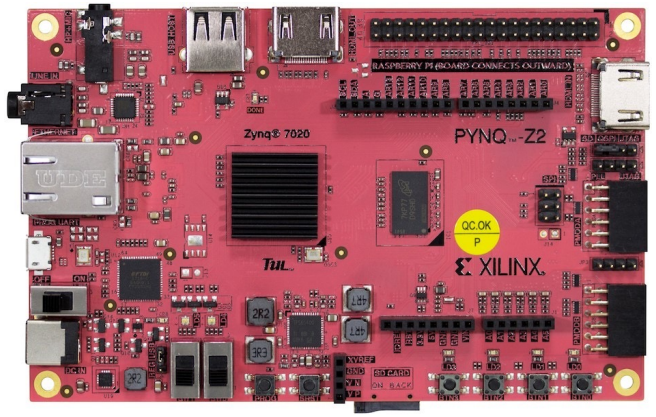
1

# 1- Introduction

- The **PYNQ-Z2** is a development board designed for embedded systems and digital design projects. The PYNQ-Z2 board features a Xilinx Zynq-7000 system-on-chip (SoC), which combines a dual-core ARM Cortex-A9 processor with programmable logic (FPGA).

- To enhance the performance, it's essential to incorporate an independently developed block, known as a **Hardware Accelerator**.

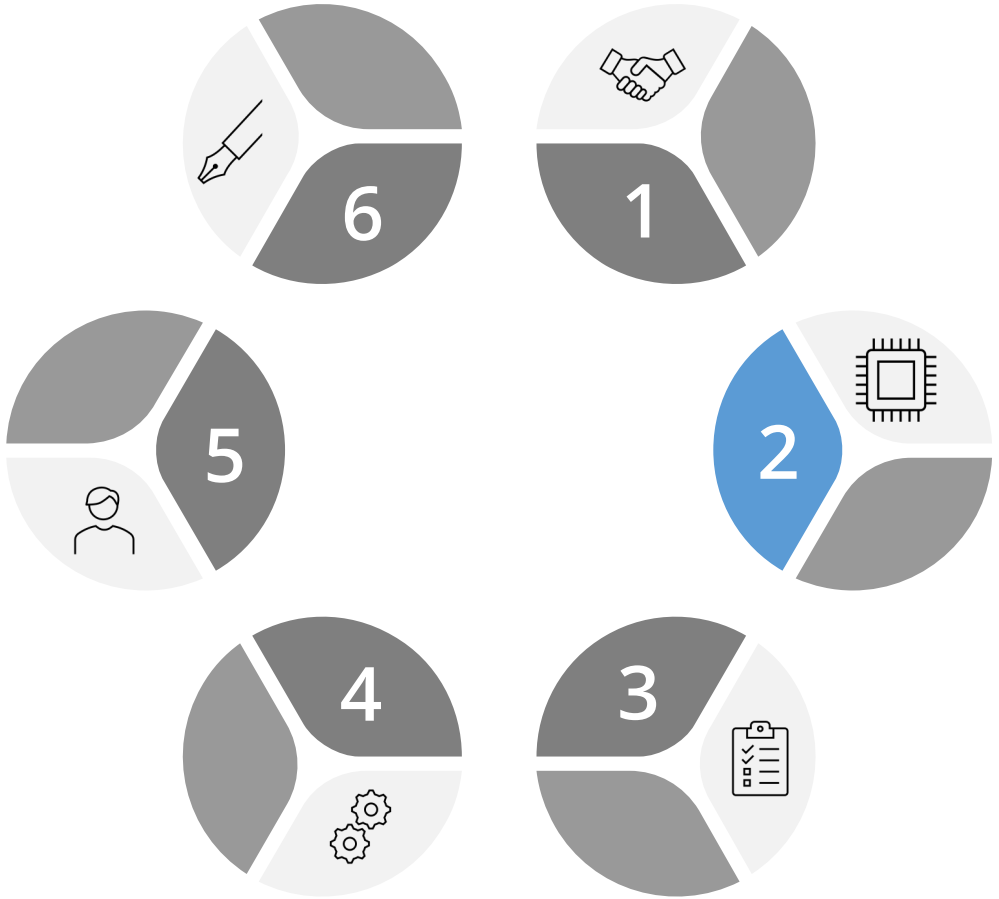
- To connect this block with the CPU of the board an open standard  is used developed directly by ARM called **AXI4 Interface**.



# 1- Introduction

- VIVADO and VITIS are used for the success of the following steps:

1. The software **VIVADO** creates the working hardware platform. Is possible specify the card on which you want to load the project. Then the VHDL code is written for the block to be inserted and through the program perform the union of the structures.
2. Through **VITIS** it is possible to connect the hardware component with the CPU through the Bare-Metal CODE. Bare-metal programming refers to programming that interacts with a system at the hardware level, without being filtered by various levels of abstraction.



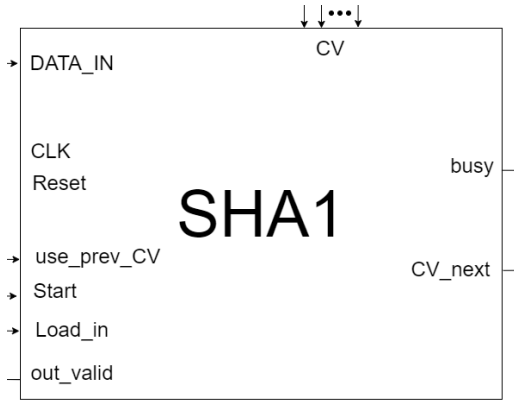

## 2- Hardware Description

The **hash algorithm** is performed on the FPGA.

# 2- Hardware Description

- The **crypto core** that perform the SHA1 algorithm is obtained from the Open Core community (https://opencores.org/projects/sha1).

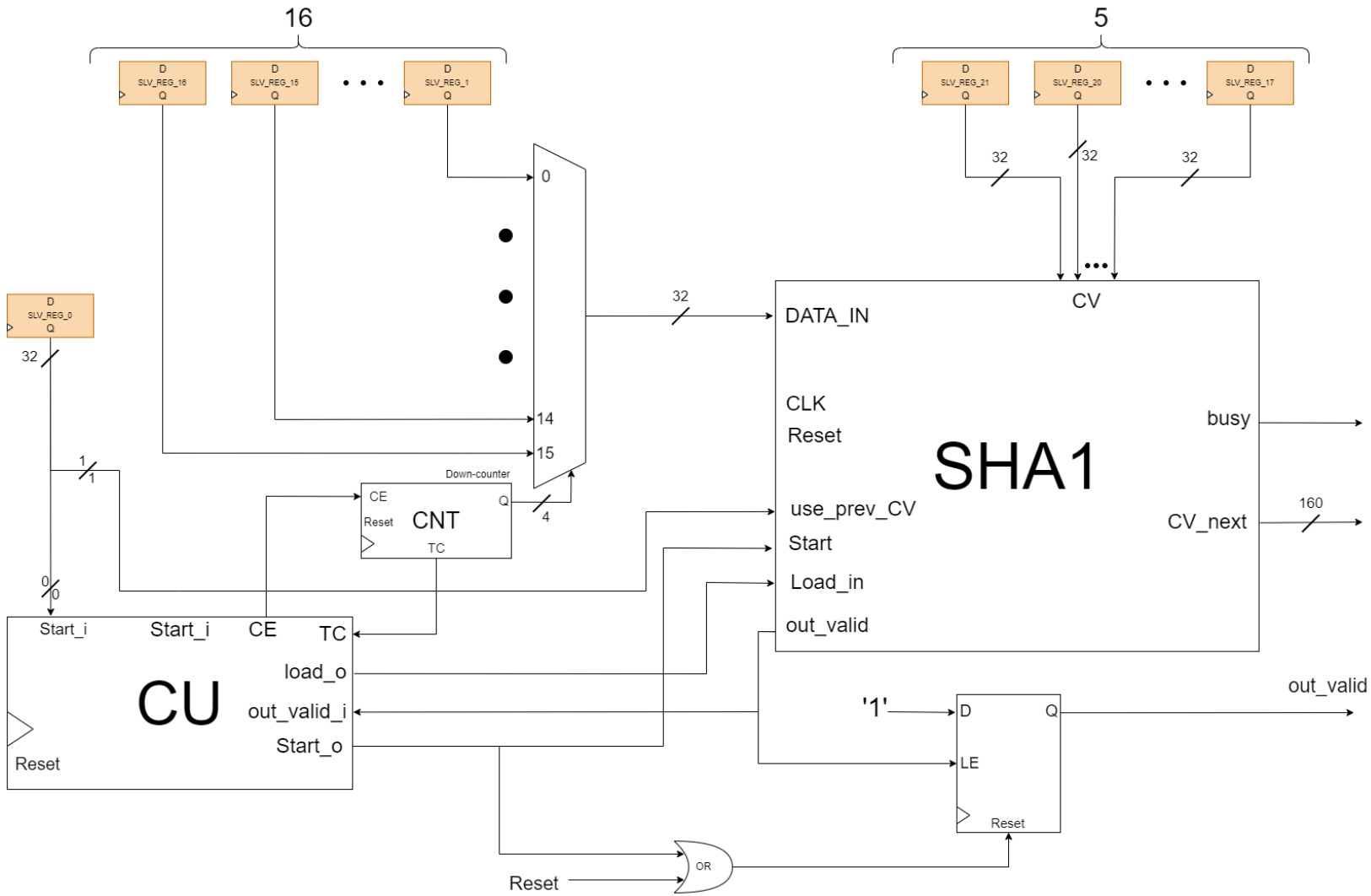
- The core works at the speed of the clock of the FPGA and generate output signals that last only one clock cycle.
  To manage in a better way the core, an hardware **interface** wraps it and handles the input/output signals for the CPU.



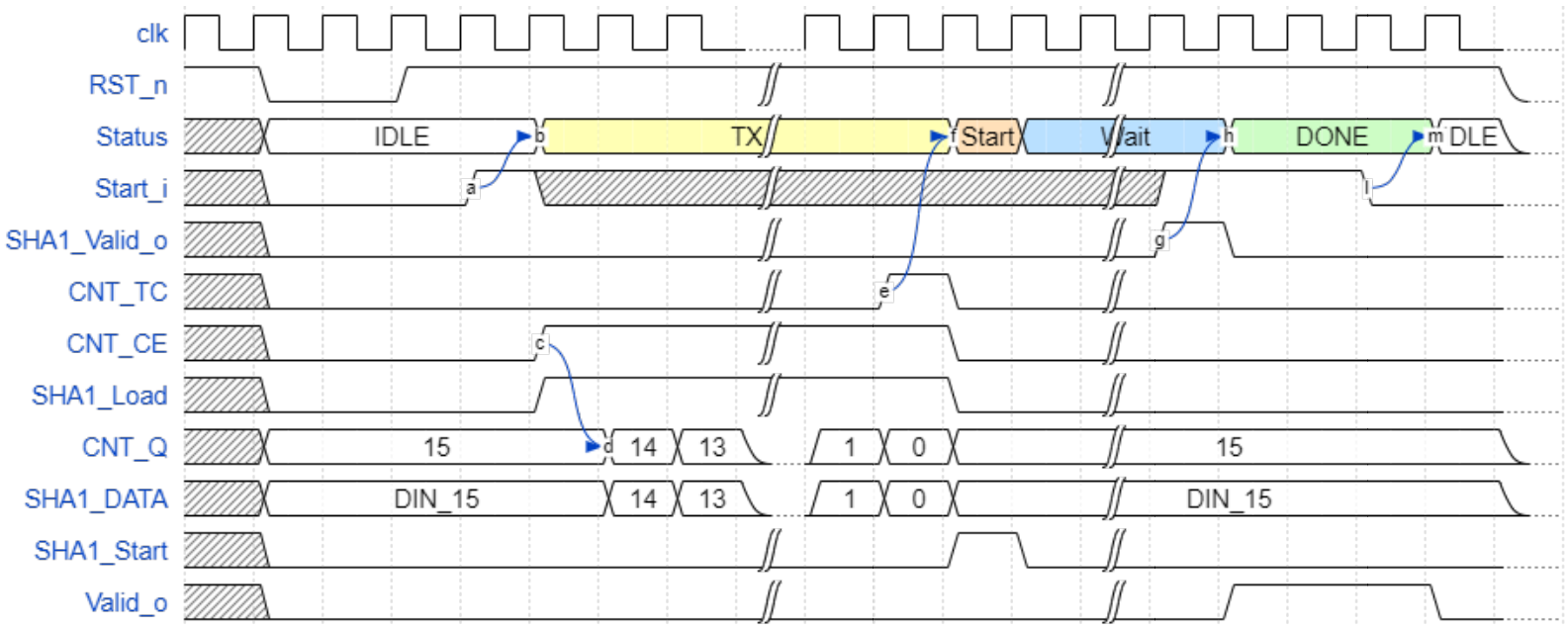
# 2- Hardware Description

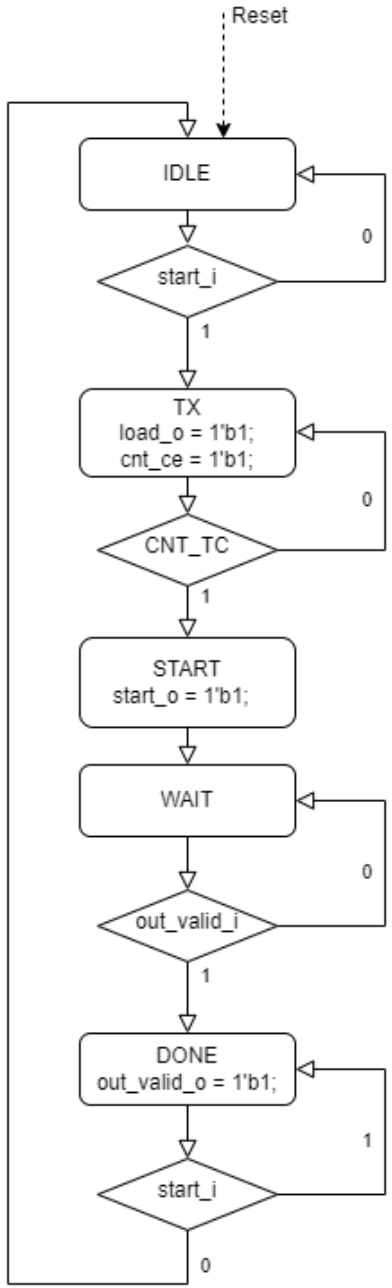
The main component of the hardware accelerator are:

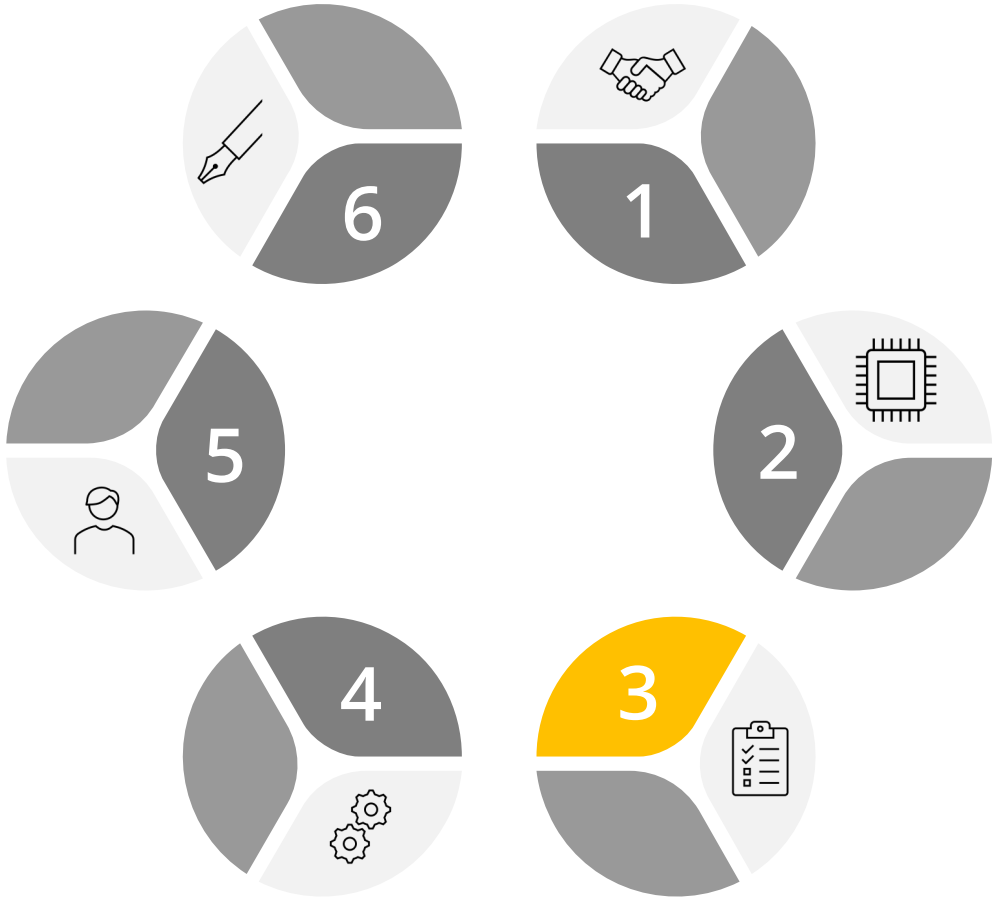
- **CU**: a FSM that manages the whole interface
- **Registers**: hold the data in input and the first CV
- **CNT and MUX**: are used to send in the correct order the input data to the core



# 2- Hardware Description

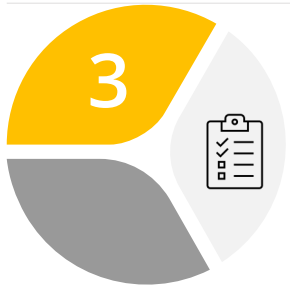





To obtain a working project, specific steps must be followed.

## 3- Work Steps



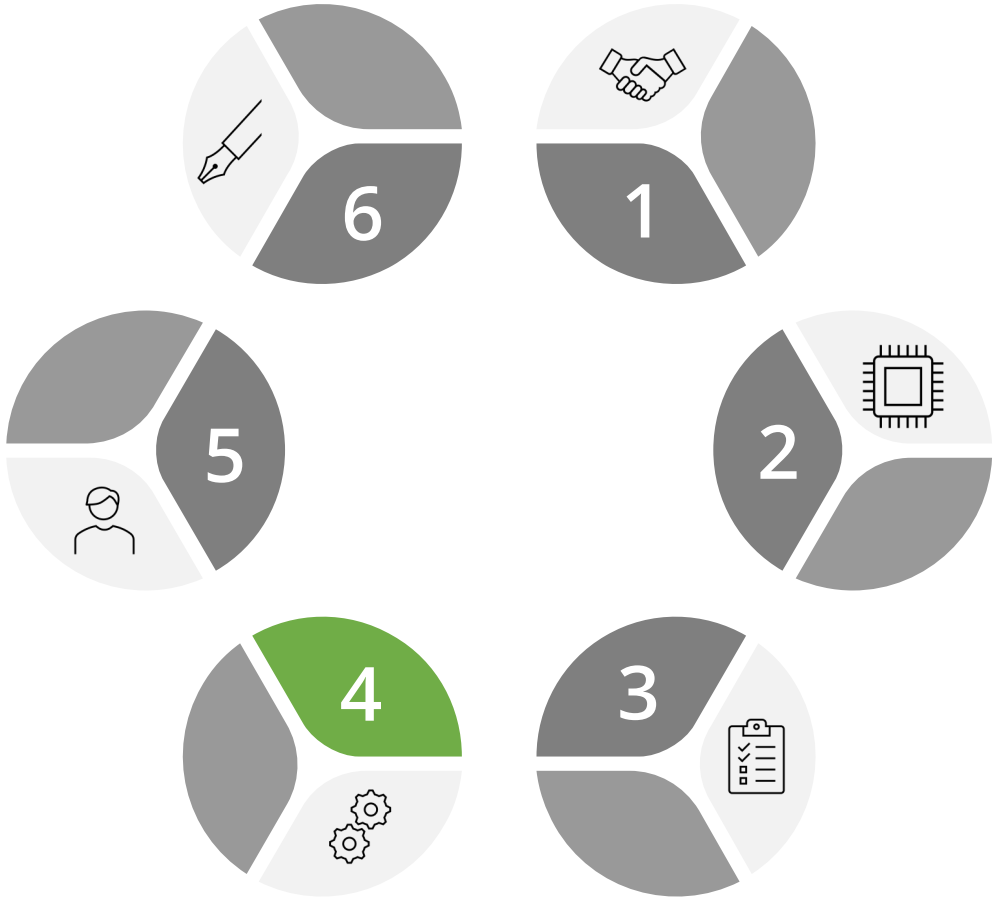

# 3- Work Steps

The remaining steps to the completion of the project are:

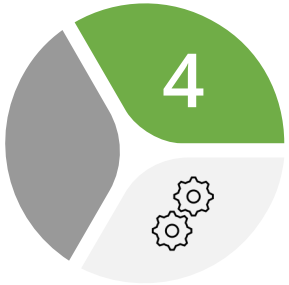
- Synthesize the accelerator for the FPGA mounted on board of the PYNQ-Z using VIVADO.
- Use VITIS to generate the interface between the FPGA and the CPU by using bare-metal code.
- Implement the driver function to work with the VFS
- Generate the kernel object with PetaLinux
- Write the user application.
- Compile and then load on the board

Without **drivers**, the OS wouldn't understand how to operate the devices. They serve as **translators**, converting software commands into hardware actions, ensuring smooth interaction between the two.

## 4- Driver Application



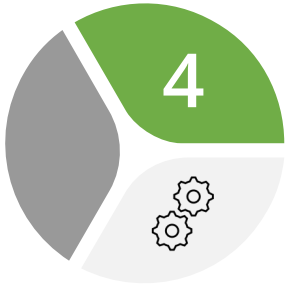
# 4- Driver Application




The driver's files are **sha1_driver.h** and **sha1_driver.c**
The implemented functions are:

- sha1_open
- sha1_read
- sha1_write
- sha1_ioctl
- sha1_release

# 4- Driver Application




- **sha1_open**
- sha1_read
- sha1_write
- sha1_ioctl
- sha1_release

This is called whenever a process attempts to open the device file.
The hardware is initialized.
Parameters:
**inode**:  a pointer to an inode object (defined in linux/fs.h)
**filep**: a pointer to a file object (defined in linux/fs.h)

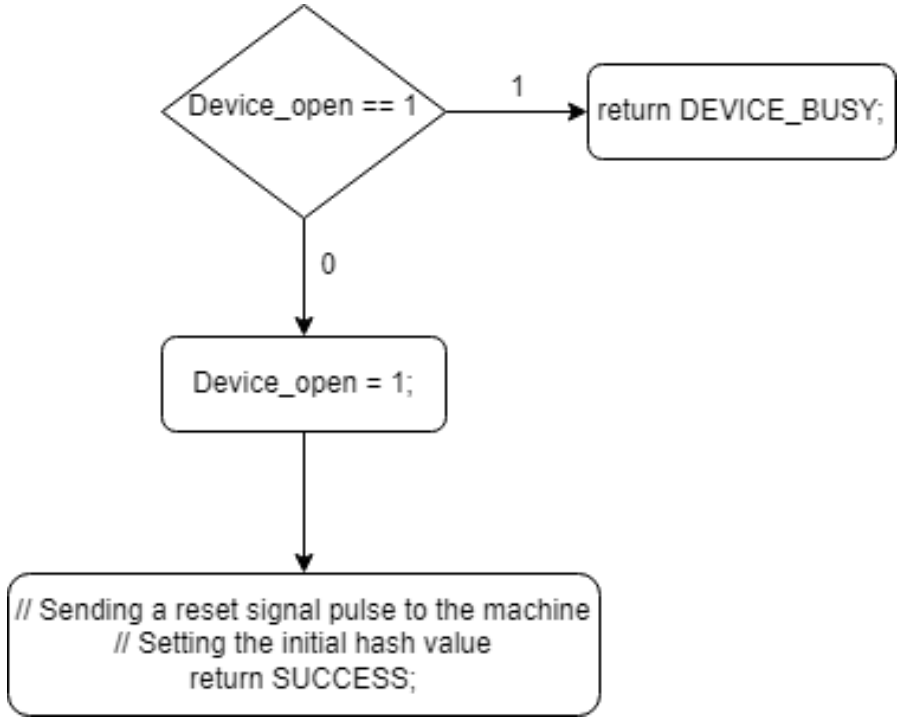
**return**: returns 0 if successful

```
static int sha1_open(struct inode *inode, struct file *file);
```

# 4- Driver Application

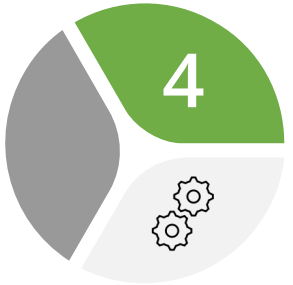


- **sha1_open**
- sha1_read
- sha1_write
- sha1_ioctl
- sha1_release




```
static int sha1_open(struct inode *inode, struct file *file);
```

# 4- Driver Application




- sha1_open
- **sha1_read**
- sha1_write
- sha1_ioctl
- sha1_release

This function is called whenever a process which has already opened the device file attempts to read from it.
Starting from the address selected using the ioctl the device registers are readed for the length of the buffer specified as function argument.
Parameters:
**file**: a pointer to a file object (defined in linux/fs.h)
**buffer**: the pointer to the buffer to which this function writes the data
**length**: the length of the buffer
**offset**: the offset if required

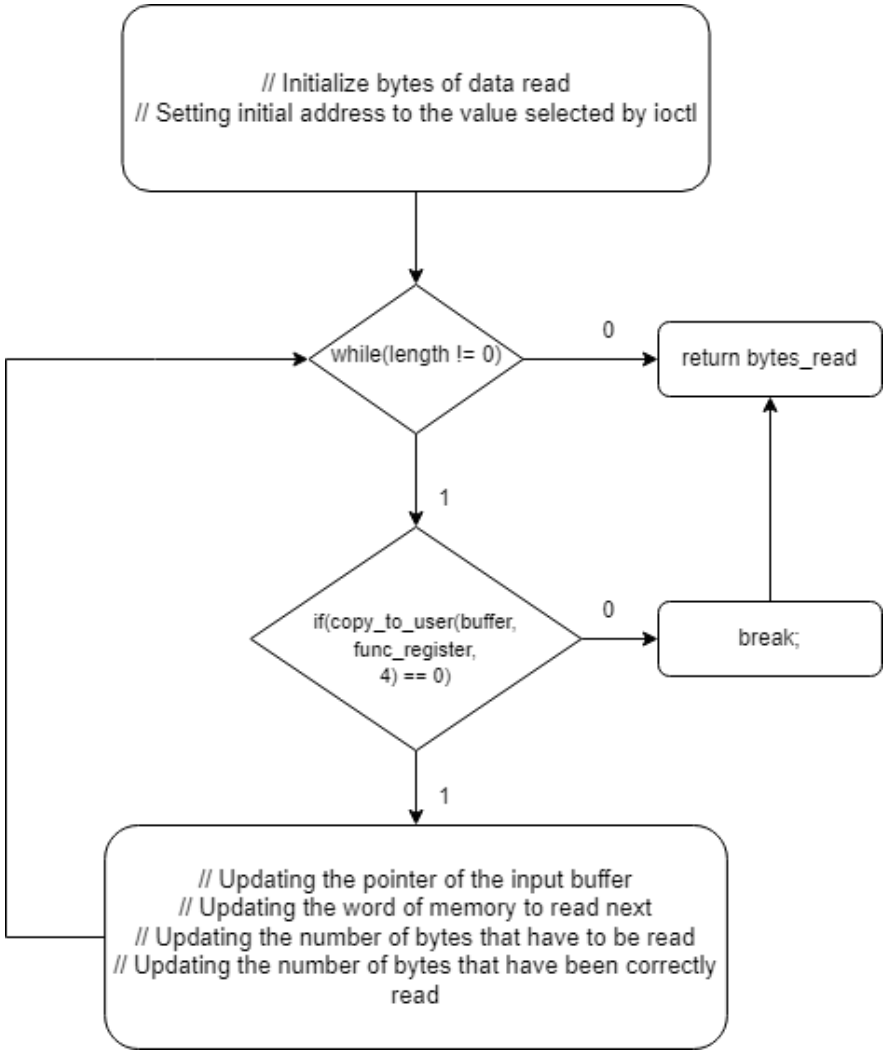
**return:** returns the number of bytes that have been correctly readed

```
static ssize_t sha1_read(struct file *file, char __user *buffer, size_t length, loff_t *offset);
```

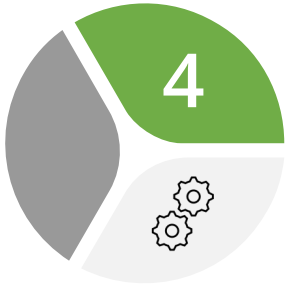
# 4- Driver Application



- sha1_open
- **sha1_read**
- sha1_write
- sha1_ioctl
- sha1_release



# 4- Driver Application




- sha1_open
- sha1_read
- **sha1_write**
- sha1_ioctl
- sha1_release

This function is called when somebody tries to write into our device file. Starting from the address selected using the ioctl the device registers are writen for the length of the buffer specified as function argument.
Parameters:
**file:** a pointer to a file object
**buffer:** the buffer to that contains the string to write to the device
**length:** the length of the array of data that is being passed in the const char buffer
**offset:** the offset if required

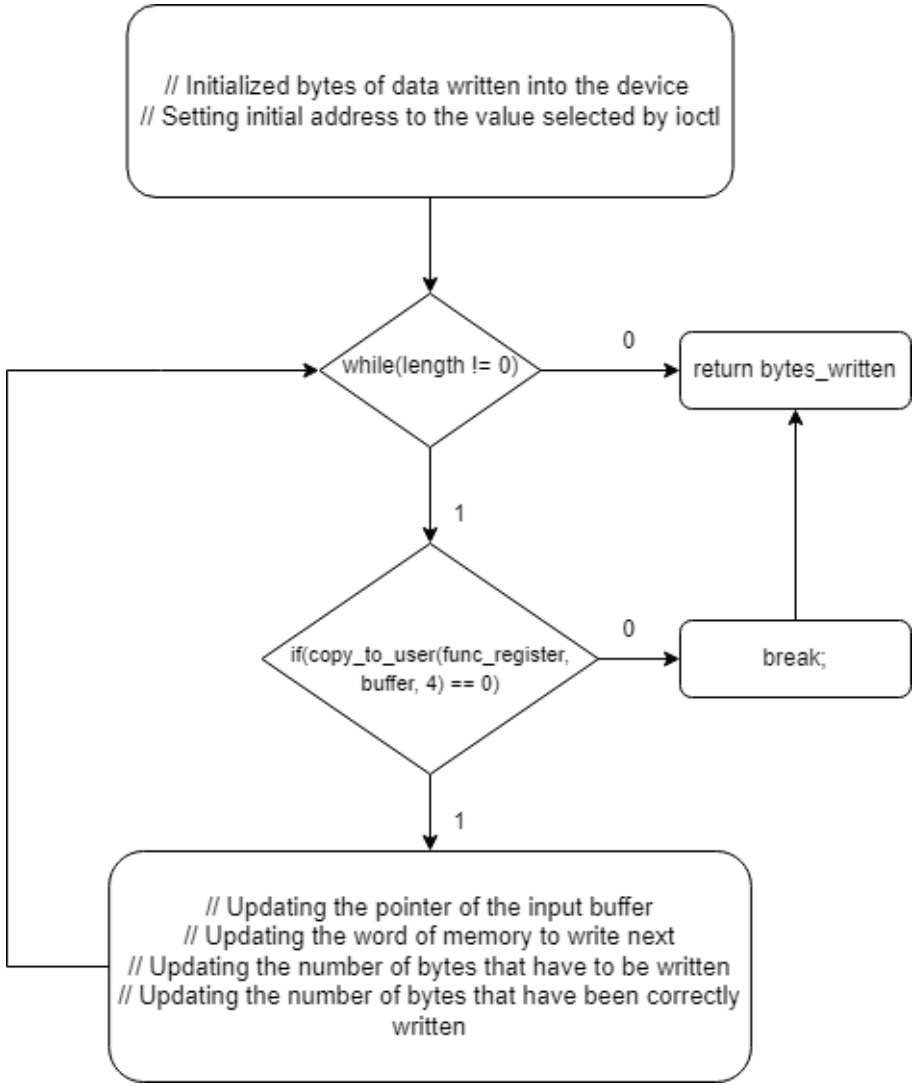
**return:** returns the number of bytes that have been correctly written

```
static ssize_t sha1_write(struct file *file, const char __user *buffer, size_t length, loff_t *offset);
```

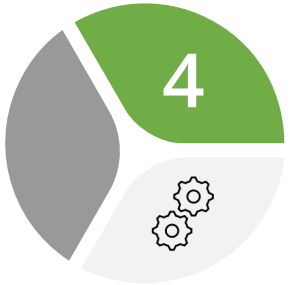
# 4- Driver Application



- sha1_open
- sha1_read
- **sha1_write**
- sha1_ioctl
- sha1_release



# 4- Driver Application



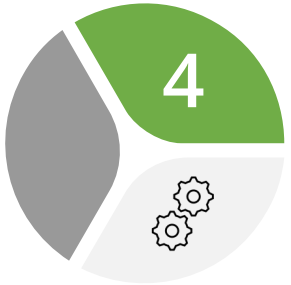

- sha1_open
- sha1_read
- sha1_write
- **sha1_ioctl**
- sha1_release

This function provides commands to the device.
Using this function it is possible to select a register to target for the read/write operations.
Parameters:
**file**: a pointer to a file object (defined in linux/fs.h)
**ioctl_num**: a command
**ioctl_param**: the arguments for the command

**return**: returns 0 if successful

```
long sha1_ioctl(struct file *file, unsigned int ioctl_num, unsigned long ioctl_param);
```

# 4- Driver Application




- sha1_open
- sha1_read
- sha1_write
- sha1_ioctl
- **sha1_release**

This function is called whenever a process attempts to close the device file.
The hardware is resetted.
Parameters:
**inode**: a pointer to an inode object (defined in linux/fs.h)
**filep**: a pointer to a file object (defined in linux/fs.h)

**return**: returns 0 if successful

```
static int sha1_release(struct inode *inode, struct file *file);
```

A program that can use the driver functions to perform a more custom algorithm and obtain an application ready for the user

## 5- User Application

# 5- User Application


A program that **elaborates the data** to make it ready for the crypto core is needed. The algorithm to interface with the user and the driver functions is contained in the files:

- main.c
- sha1_lib.c
- sha1_lib.h

**main.c**:

- The user can decide to perform the hash algorithm on a string or a file. On the user interface is shown the result of the sha1 algorithm.

# 5- User Application


**sha1_lib.c** and **sha1_lib.h**:

- These files handle the management of the driver device by using the driver's functions and checking for any errors that might occur.

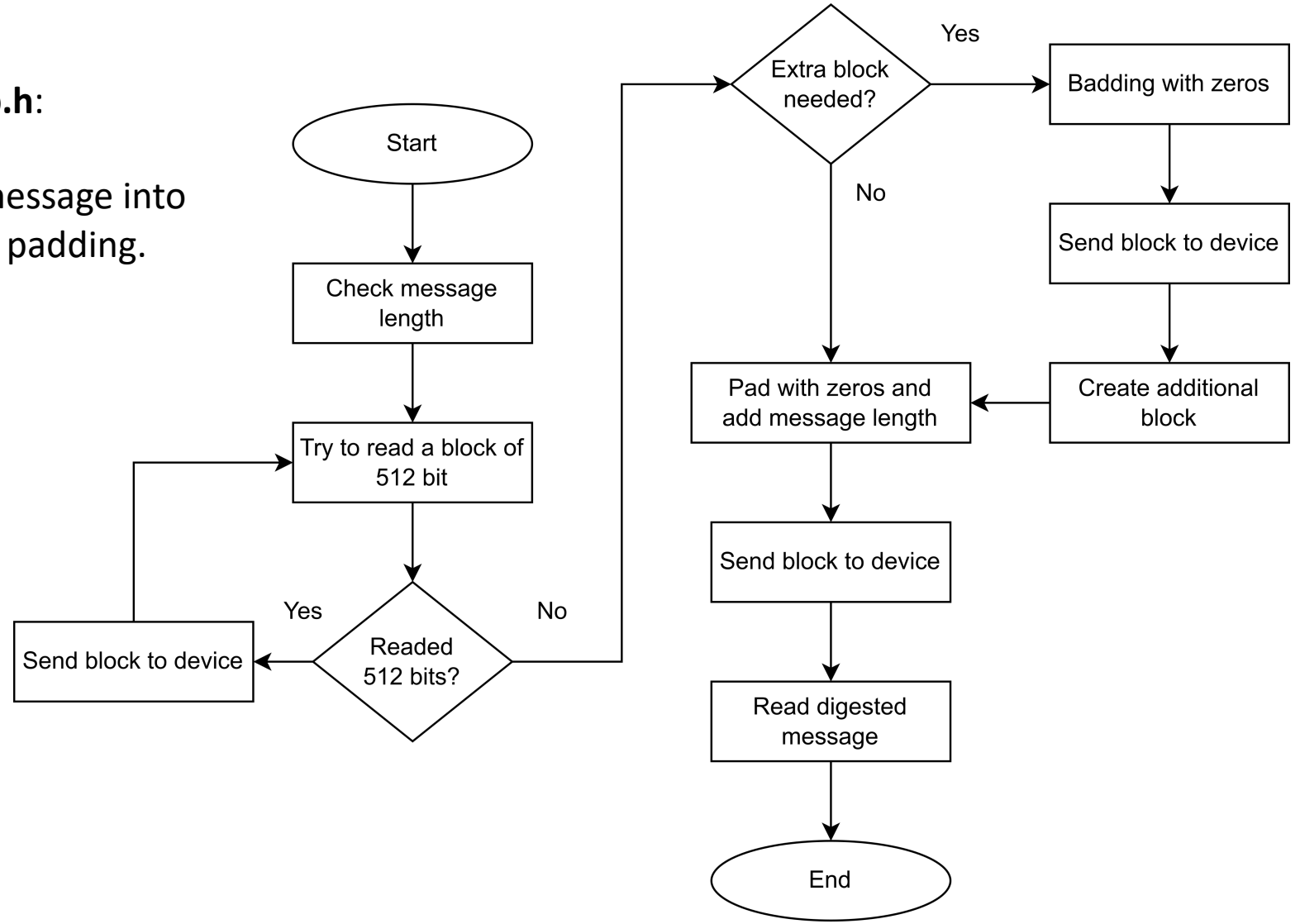
- Firstly, the length of the message is observed and the bytes are counted. Subsequently, the message is hashed using the original **initialization vector** (CV). The content of the first block (512 bits) is read. The process continues with the next block, using the CV created from the previous iteration. At this stage, it is checked whether the last read block requires an additional block for padding. Finally, **padding** is performed by inserting the right sequence of bits.

# 5- User Application



**sha1_lib.c** and **sha1_lib.h**:

Algorithm to split the message into blocks and perform the padding.

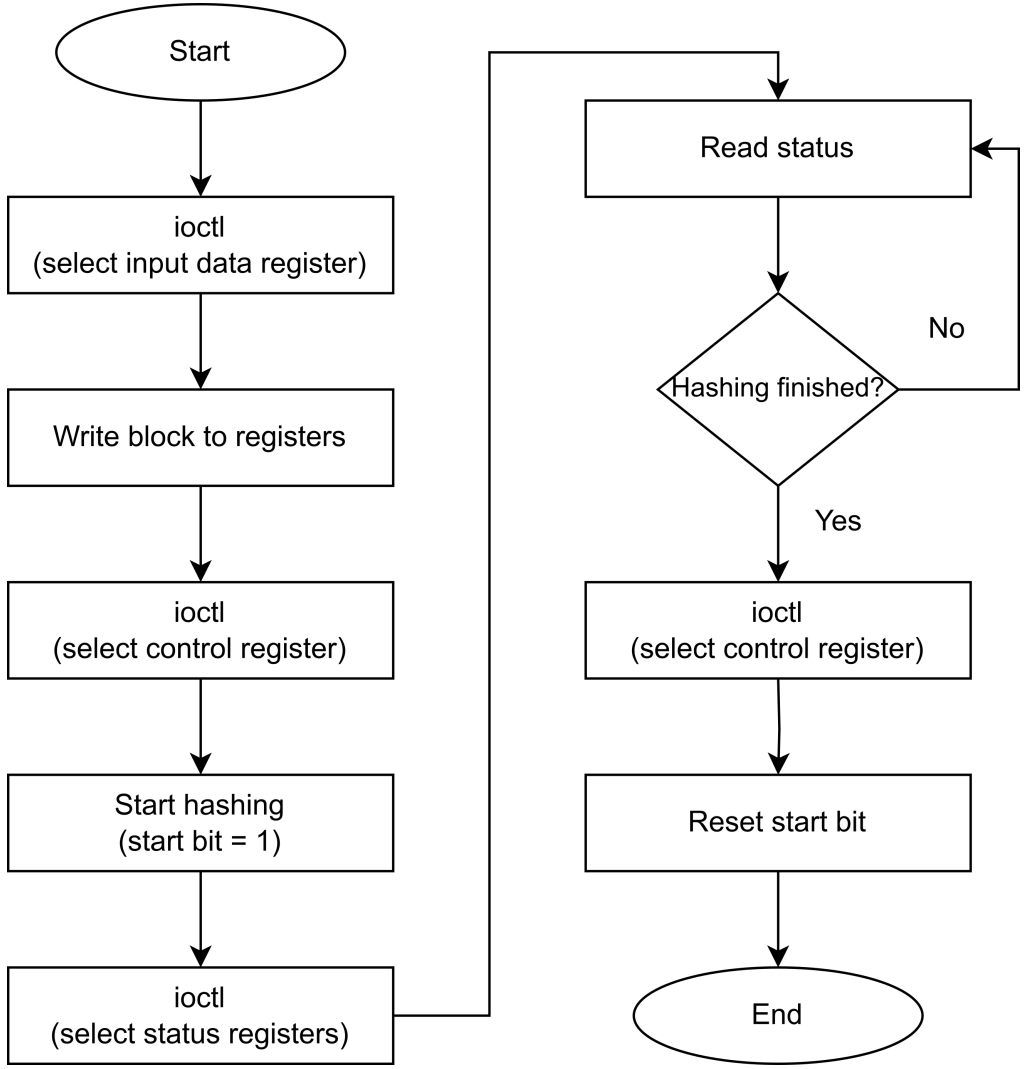


# 5- User Application



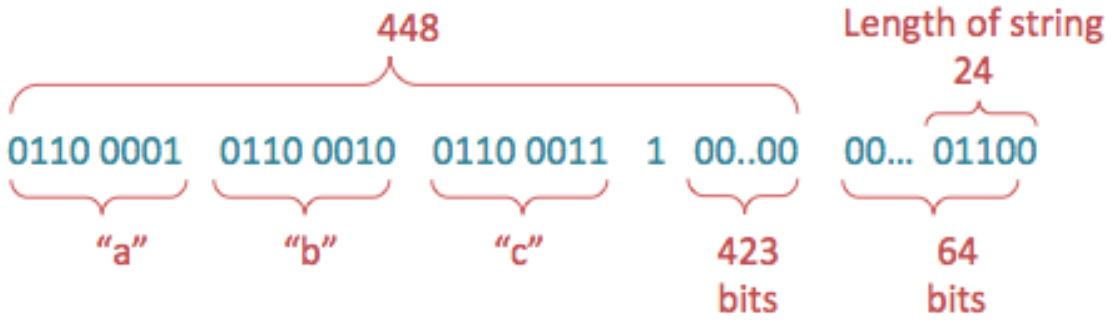
**sha1_lib.c** and **sha1_lib.h**:

Algorithm to send the data to the FPGA.



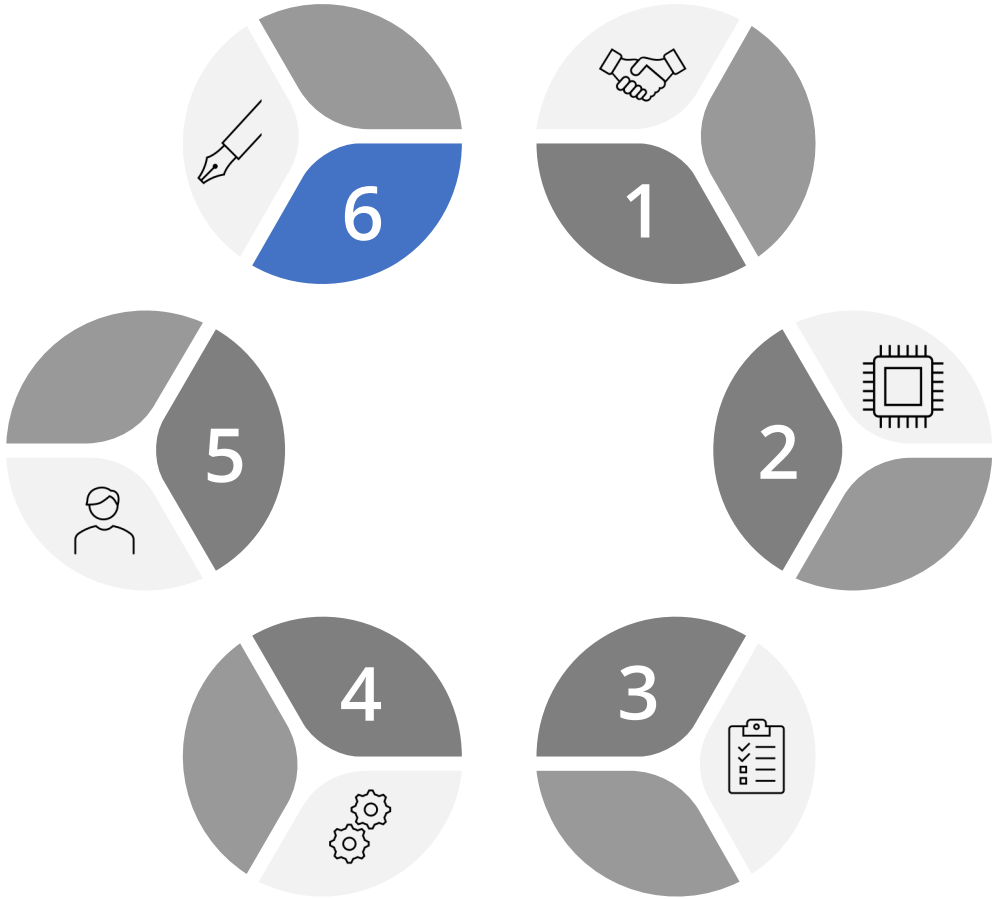
# 5- User Application


- The **padding** is obtained by appending a 1, followed by enough 0s until the message is 448 bits. Then the length of the message represented by 64 bits is added to the end, producing a message that is 512 bits long.
  If the last block is longer than 448 bits, then one another block is used to be sure to be able to add the last 64 bits.



# 6- Final Considerations

Is this project really working?
How much is this hashing algorithm safe?
How the source-code can be obtained?



# 6- Final Considerations


Example of the **successful** running of the project:

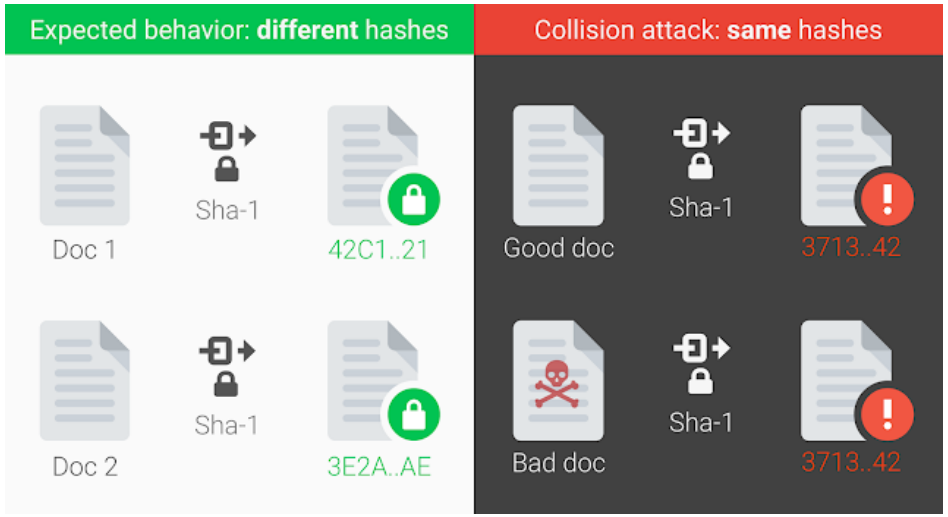
```
PetaLinux 2022.2_release_S10071807 sha1 /dev/ttyPS0

sha1 login:
```

# 6- Final Considerations

6

Starting in 2005, the **security** of the SHA-1 algorithm **became uncertain**, leading many organizations to seek a replacement. In 2017, researchers from CWI Amsterdam and Google reported a successful collision attack against SHA-1.
The initial attack focused on a simplified version of the algorithm and managed to crack **53 out of 80** rounds using less than $2^{80}$ computational operations.

In 2015, a significant breakthrough occurred in the field of cryptography. An attack was done against the SHA-1 hash function, requiring only $2^{57}$ calculations. This was the first successful attempt to hack the full version of the algorithm. This event is widely known as the "**SHAppening**".

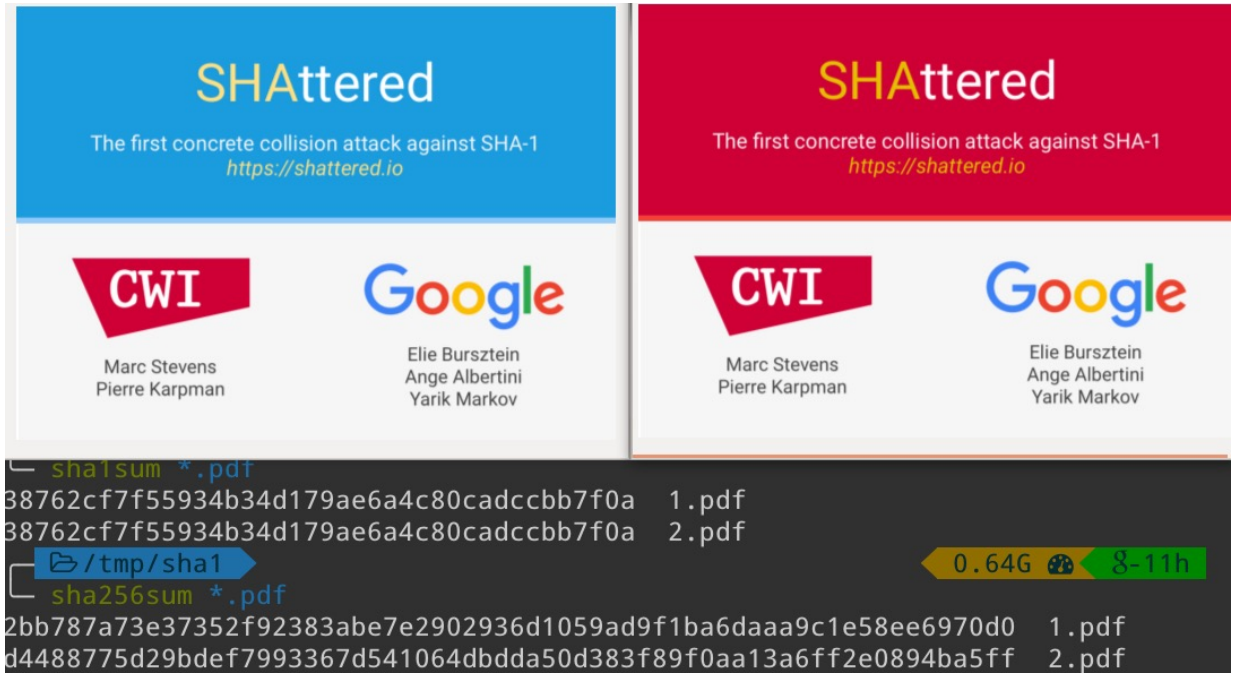


# 6- Final Considerations

6

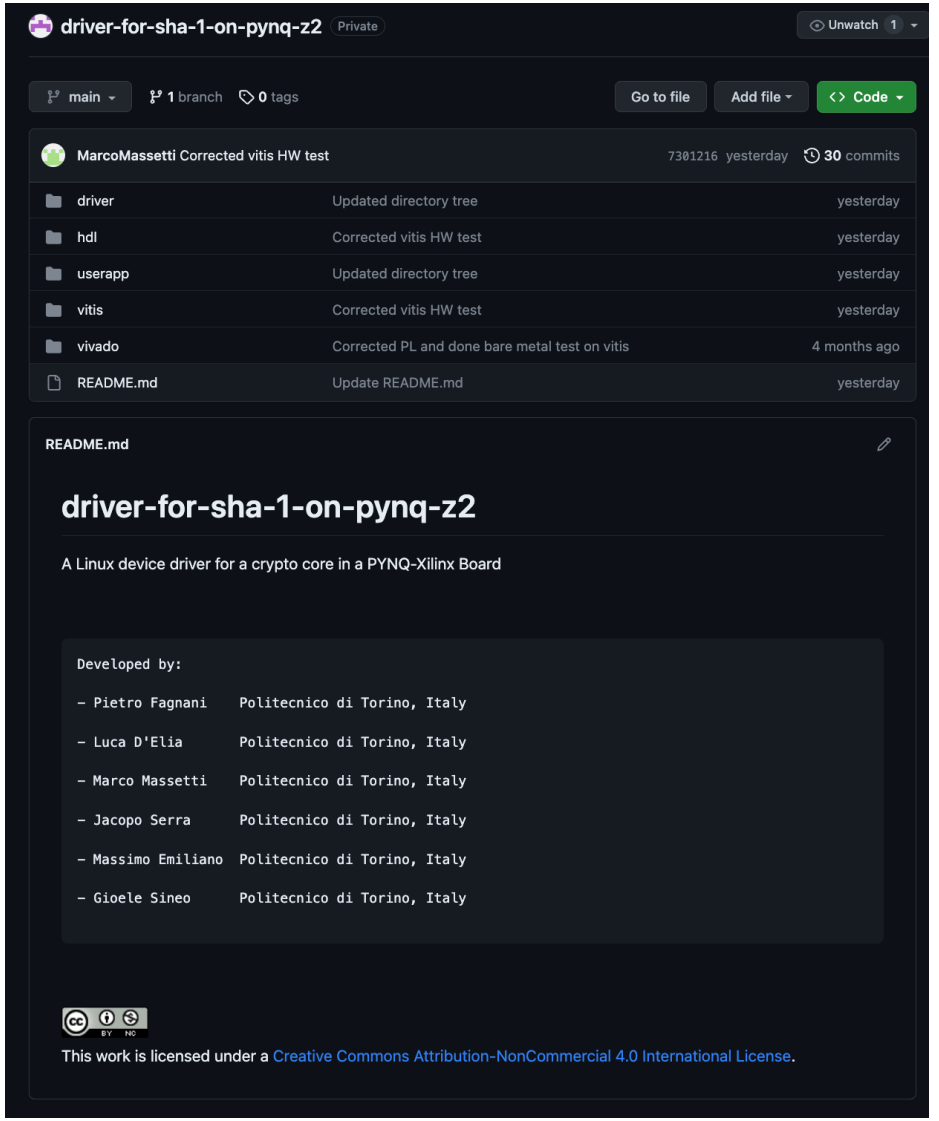
In 2017, Google generated a collision, later named “**SHAttered**”, that put a definitive end to the SHA-1 algorithm.
The company was able to generate two different .pdf files having the same SHA-1. Analyzing the two files shows that they differ from each other by only a few bytes, but despite this, the attack required an incredible amount of computation, as described in the study's official paper.

https://shattered.io/



# 6- Final Considerations


The **source code** of the project is distributed on a public [GitHUB repository] using a CC BY-NC 4.0 license.




1
2
3
4
5
6
Do you have any question?


THE END

-

THANK YOU FOR THE ATTENTION