
UNIVERSITAT DE BARCELONA

**Bachelor's Degree Final Project**

**ELECTRONICS TELECOMMUNICATIONS ENGINEERING**

**Faculty of Physics**

---

# Low power lifetime evaluation on an open hardware RISC-V system

---

Barcelona, June 16th 2024

Advisor: Dr. Angel Dieguez Barrientos

Author: Guillem Prenafeta Guilera

# Acronyms

**ALU** Arithmetic Logic Unit. , 2

**CMM** Central Mass Method. , iii, v, 10, 18, 19, 21, 22, 31

**CPU** Central Process Unit. , 33

**Die** Integrated circuit.

**FIFO** First In First Out memory. , 12, 13

**GPIO** General-Purpose Input/Output. , iii, iv, 8, 13, 33

**ISA** Instruction Set Architecture. , 1, 2

**LiDAR** Laser Imaging Detection and Ranging. , ii, v, 1, 4, 5, 7

**MCU** Microcontroller unit.

**RA** Regression Algorithm. , v, 10, 11, 18, 20–22

**RV32** 32bit RISC-V. , 12

**RV32I** RISC-V 32b instruction set.

**RV32IM** RISC-V 32b instruction set with MDU.

**SPAD** Single-Photon Avalache Diode. , ii, 1, 2, 4, 8

**SSI** Synchronous Serial Interface. , iii–v, 9, 12, 32

**TDC** Time to Digital Converter. , 17

**ToF** Time of Flight. , 1, 5

**Abstract**

Nowadays, the number of sensors present in our daily lives is growing rapidly. And because of this, large amounts of data need to be transmitted and processed. In order to avoid power wasting due to the amount of data to be transmitted outside, sometimes the best way is to process the data directly in the sensor. That is called *edge computing*.

This document will discuss a case where edge computing is applied. Specifically in the lifetime computing of a Single-Photon Avalache Diode (SPAD) sensor. The most common use of SPADs are biomedical devices for the point-of-care, and Laser Imaging Detection and Ranging (LiDAR). The main objective is to develop an *edge device* which will process all the SPAD data. As the title of the project states, a RISC-V core will be implemented with the necessary modifications in hardware and the algorithms to process all the data.

# Contents

# List of Figures



# List of Tables

# 1 Introduction

In our edge computing project, the device near to the sensor is called *edge device*, and in this project it will be based on an open source RISC-V core. RISC-V is an open standard Instruction Set Architecture (ISA) which importance lies in the fact that it is open source, which means that anyone can take the ISA and apply it in their design without having issues of copyright, patents, licenses...

Many companies are offering or have announced RISC-V hardware. Open source operating systems with RISC-V support are available, and the instruction set is supported in several popular software toolchains, which in the past these characteristics were only available in the x86 architectures which have royalties and only a few million-dollar companies have the rights to use it.

Once the edge device has been chosen and designed, the algorithms for processing the lifetime constant will be developed so a study of the SPADs sensors and its applications is required. A SPAD is a type of diode capable of generating an avalanche of carriers with only the energy of a single photon.

In many applications, it is common to measure the time between radiation events and different properties of this time. For example, in LiDAR technology the time it takes for light to be received (Time of Flight (ToF)) is used to capture spatial images of the environment. Using a SPADs sensor can be very advantageous due to its high sensitivity and low power consumption characteristics. Other applications need probabilistic light decay (time constant) to measure different physical quantities such as fluorescence or intensity itself, the use of SPADs can be also very interesting in these areas.

The intention in this work is to further reduce the data through analysing the histograms and sending data such as lifetime. However, integrating a processor would allow much more play with the data. For example, to calculate where the peak light is in a background, for LiDAR.

This project will develop the *edge device* and will implement the algorithms for measuring the lifetime constant in a exponentially decaying histogram of accumulative events measured by the sensors. The *edge device* as it was said will be a RISC-V processor which will be developed up to the logic synthesis stage remaining to do all the physical stage (which will be carried out in the future but outside the limits of this work).

# 2 Background

The background study is made of two parts. The first focuses on the edge device, the RISC-V, and the second one on the application of the device, the SPAD.

## 2.1 RISC-V Processor

The RISC-V, is a microprocessor that works with the RISC-V ISA. An ISA or Instruction Set Architecture, is part of the abstract model of a computer that defines how the CPU is controlled by the software. The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done[1]. In other words, is the language that the CPU understands, and with that language we have to program it.

The RISC-V ISA was developed in 2010 by the University of Berkeley, California. The project was developed in order to make a practical ISA that was open-sourced, usable for anyone and applicable in any hardware or software design without royalties. Another advantage that x86 architecture does not have is its modularity. It can be easily modified, and this can be very interesting due to the fact that we can add special instructions appropriate for the application.

Now the most important things of the architecture will be showed. All the RISC-V ISA information can be obtained from RISCV organization. The first step you need to do once you have chosen the RISC-V ISA is to choose which instruction set you want to work with.

- **RV32I** In this instruction set, the word is 32 bit length and the register file[2] have 32 registers of 32 bits each one.
- **RV32E** Is very similar to the RV32I, the difference is that the register file have 16 registers. This can be appropriate for applications that we do not need a lot of computation and the size is important.
- **RV64I** This instruction set works with words of 64 bit length, also it have some instructions in order to work with LSB.
- **RV128I** It is the same as RV64I but know each word have 128 bits.

Once chosen the instruction set that fits the system you can add different common extensions. However, you can modify your ISA to fit a knew extension for your application. In our system a Multiplication and Division module is added in order to make a faster algorithm. Despite more area is required, we avoid making loops which will cost time and energy.

[1] Definition obtained from **ARM** website. link

[2] The **register file** of a processor is a register array which stores data between the different blocks of the CPU, for example the Arithmetic Logic Unit (ALU), and the memory.

Table 1: Most common RISC-V ISA extensions (source link)

| Abreviation | Functionality |
|---|---|
| **M** | Integer Multiplication and Division |
| **A** | Atomic instructions |
| **F** | Single precision floating point |
| **D** | Double precision floating point |
| **Q** | Quad precision |
| **L** | Decimal floating point |
| **C** | Compressed instructions (16 bit) |
| **B** | Bit manipulation |
| **J** | Dynamically translated languages |
| **T** | Transactional memory |
| **P** | Packed SIMD instructions |
| **V** | Vector operations |
| **N** | User level interrupts |
| **H** | Hypervisor |

## 2.2 SPADs

Now we are going to focus on the SPAD and its applications. First, a brief explanation of a SPAD sensor will be given to see its main characteristics and then its different applications in different fields will be explained.

A SPAD or **S**ingle-**P**hoton **A**valanche **D**iode, is a P-N junction photodiode biased above the breakdown. It was achieved in CMOS in 2003 thanks to the creation of a Geiger mode APD (SPAD) in a high voltage $0.8\mu m$ process[2].

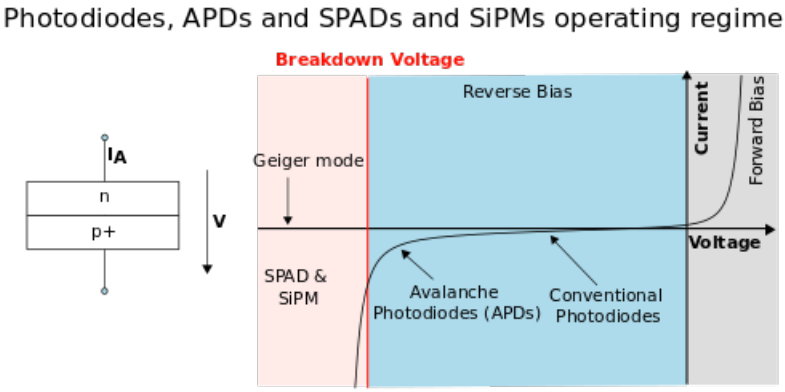



Figure 1: Operation Regimes of a solid state p-n junction including conventional photodiodes, avalanche photodiodes (APD) and and single-photon avalanche diode (SPAD) or SiPMs source link

The SPAD is operated in the GEIGER zone, where the p-n junction is strongly reverse-biased, where the electric field is above critical (3x$10^5$ $V/cm$ in silicon[3]), and with the simple energy of a photon (impact ionization) it can be generated an avalanche current resulting in a pulse of a few $\mu A$ limited by the load resistance.

Its main advantages are its high sensitivity of the single photon detection, high gain, high speed response, insensitive to ionizing radiation and magnetic fields [4].

These characteristics can be very interesting when you want to measure with precision the time between and event and a photon hit[3]. Some applications of the SPAD sensor are listed bellow and will be explained in more detail.

- **Positron emission tomography**
- **LiDAR**
- **Fluorescence**
- **Cameras**
- **Quantum computing**

In quite a few applications the **ToF**[3] of the light is used to take pictures.

One example is the Positron emission tomography (PET), a imagine technique used in Nuclear medicine to find tumors and the search of metastases [5].

An other example can be in LiDAR applications, where a photon source like a LASER emits photons in order to compute its ToF to calculate the distance. Thus can result in a 3D image of the space very useful for applications where the robot/AI needs to interact with the space. For example in autonomous cars that's very important, or for autopilot drones.

Figure 2: Image obtained from LiDAR sensor in autonomous driving. (provided by Velodyne LiDAR

Another application where the time takes an important place is for fluorescence experiments. Where there is a magnitude called lifetime constant that describes how the light emitted by a material has a determinate lifetime. This can be measured by a SPAD sensor making a kind of histogram[6].

Finally another interesting application to mention is the measure of light intensity as a alternative way to CMOS pixels [7]. If there is a light source emitting photons at a certain intensity, it can be defined a photon flux $\Phi$. Intrinsically that's a Poisson distribution and yo can make an exponential probability histogram with the photon hits. So measuring the time (lifetime constant) you are directly measuring the inverse of the light intensity.

In the next figure provided by *Canon*, they are showing its SPADs camera for night vision and its advantages versus conventional CMOS. Unlike conventional CMOS cameras, a SPAD camera is highly sensitive due to its characteristics.

[3]The ToF or Time of Flight technique uses the time it takes for the photon to travel through space to calculate the length of it by knowing the propagation velocity in the medium.

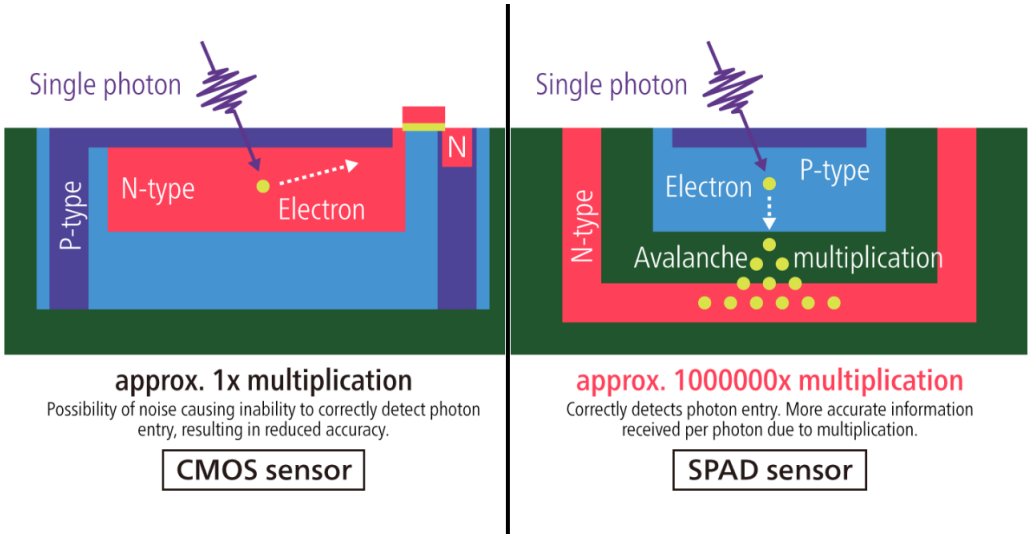



Figure 3: CMOS and SPAD detector comparison provided by Canon

Finally an emerging technology where SPAD can be implemented, is in quantum computing. In this discipline the bit becomes into the particle quantum states and measuring this transitions is one of the main objectives, capturing the emitted photons can be easily made by a SPAD[8].

So as we have seen, the SPADs sensor has a wide range of applications, and due to the fact that it can be integrated in CMOS (solid state p-n photodiode), we can take profit from it putting in the same die the processing unit (edge device).

# 3 Market Analysis

The SPAD market is poised for substantial growth in the coming years through its current and emerging new applications.

We can divide the market with those companies that manufacture the SPADs (those one's that have foundries and are able to implement the semiconductors) and the other ones that use the technology to apply it for different applications.

- **Sony, Canon...** Optical imaging with SPADs
- **Adaps Photonics** Company that uses the SPADs for ToF
- **Toshiba** Quantum light detection using SPADs
- **STMicroelectronics** ToF sensors using SPADs
- **TSMC, XFAB, Onsemi...** SPADs manufacturers

However, many large companies are already investing in this technology due to its characteristics. In addition, various start-ups are emerging to satisfy different application possibilities.

The SPAD market was US$412 million in 2022 and is expected to reach US$879 million in 2029[9].

The integration of SPADs with CMOS electronics began in 20032.2. Since then they have been integrated with timer counters (TDC) taking the information of each hit outside the chip[10], with counting circuits, with circuits that make histograms, etc. Obviously, the consumption associated with the transfer of each hit to the outside is the highest. In the case where the lifetime constant is to be calculated, on-chip histogram generation reduces the data sent, from millions of captures to the number of bins by the number of bits in each bin, this will be reviewed in the power analysis section (5.5). The intention in this work as it was said is to further reduce the data through analysing the histograms and sending data such as lifetime. However, integrating a processor would allow much more play with the data. For example, to calculate where the peak light is in a background, for LiDAR.

To date, there is no spad integrated with a processor, which opens up new possibilities in many markets. In addition, the project could be interesting for those companies that are already implementing SPAD technologies and are searching ways to reduce the power and increase the computation capacity.

# 4 Conception Engineering

## 4.1 Proposed solutions for the hardware

All the project is focused on taking the SPAD module and join it with the RISC-V microprocessor in order to do data processing.

The SPAD module was developed by the Electronic and Biomedical Department of the University of Barcelona[11] and it have a register file as a means to control it. The processor should control this module and be able to communicate with the outside.The block diagram of what it have to be implemented can be seen in the following figure.

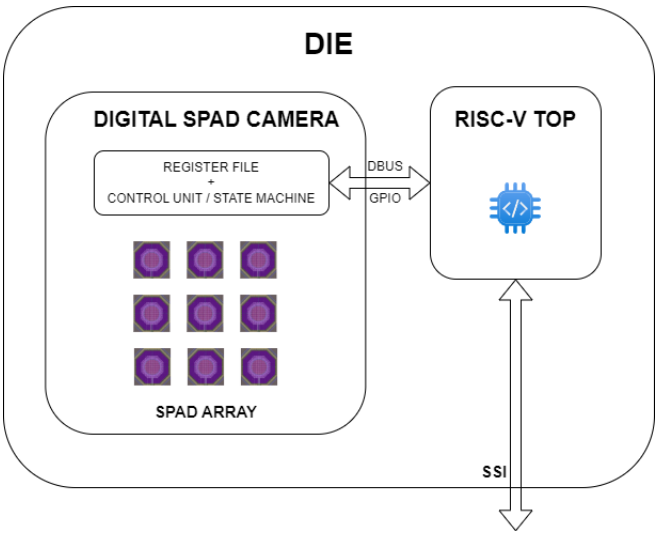



Figure 4: Block diagram about the TOP CHIP

The main block is the chip die. Inside it there are two parts, the SPAD camera module and the RISC-V module which we have added.

So we will focus on the RISC-V TOP module which will control the SPAD camera/array. This block can be divided by the RISC-V CPU and the peripherals.

In order to get the RISC-V core, different open source designs were reviewed such as the **PicoRV32**[12] a size optimized 32b cpu. But finally the **ServRISCV**[1] was chosen due to its tiny size. All of them are completely open source in case one day they reach the stage of manufacture and sale.

Once the RISC-V core is chosen, the peripheral blocks need to be developed. In order to run properly a program, a memory for instructions and data is needed so a register set had to be implemented which in the future will be probably an IP memory block provided by the foundry (depend on the implementation the memory changes, if it is in a FPGA a register file is used, and in the synthesis a memory IP is used).

Then for the communication with the exterior different communication protocols were revised. Finally the chosen one was a synchronous serial interface for its simplicity and robustness. Apart from designing the communication interface a standard GPIO module was designed to manage interrupts and values with the SPAD module and the exterior. All this modules capable of generating interrupts controlled by an interrupt module.

The memory of the RISC is volatile, because we want speed and small size, however, at a later stage it could be discussed to add some kind of non-volatile memory such as a FLASH to store the program.

A module for programming the instruction memory is also required. This is designed to receive the instructions by SSI which are then written to the instruction memory.

Finally, a timer was designed in case a timer module is needed (if we want to use timer interrupts). In the next figure (5) the block diagram of RISC-V module can be seen with its respective modules. (All the modules can be found in the *github* repository [13]).

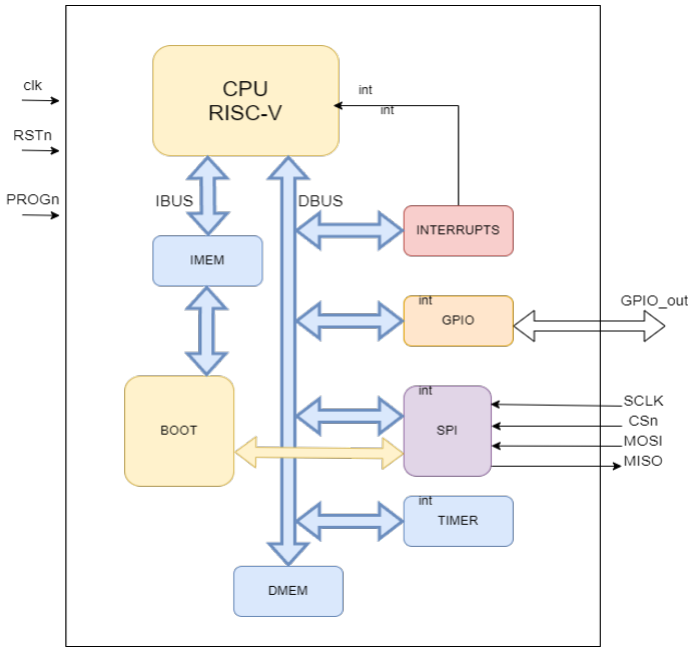



Figure 5: RISC-V TOP block diagram

In the figure above (5) we can see the different blocks. On the top of the figure there's de RISC-V CPU which inputs/outputs are the data bus (DBUS), the instruction bus (IBUS), the timer interrupt input and the reset and clock inputs. The register file is already inside the CPU and have de 32 registers of the RISC-V ISA plus a set of registers that contain CRC values. In the detail engineering section the module will be discussed with more detail (5.1.1). Then connected to the instruction bus we have the instruction memory and the bootloader which will use it to program it.

Finally there are the different modules connected to the data bus, the Interrupt module, the GPIO module, the SSI (SPI) module, the Timer module and the data memory.

## 4.2 Proposed algorithms to obtain the Lifetime constant

In this section different algorithms for our application will be revised. The main idea is to take advantage of the programability of the processor to implement different options.

The main two algorithms are the **CMM** algorithm which computes a simple sum of the values and a larger algorithm which computes the regression of the data. I will call it **RA**. It may be possible in the future to take advantage of the processor's ability to implement a neural network in software to process lifetime. But it would also be possible to program a hardware module that implements this network and use an instruction from the processor to activate it. This is a clear advantage of using an open source processor where the ISA can be modified.

### 4.2.1 CMM

The CMM algorithm starts with the concept of an exponential probability distribution function. As the aim is to find the lifetime constant we will use as an example the application of seeing the light intensity measuring the lifetime constant.
Assuming a constant photon flux $\Phi = \frac{dN}{dt}$ where $N(t)$ is the photon hits integer function in time. By definition that's a Poisson distribution where the probability to have 1 or more photon hits is

$$P(t < t_i) = 1 - e^{-\frac{-t}{\tau}} \tag{1}$$

And if we calculate the expected value of this distribution ($< t >= \int \frac{dP}{dt} t dt$) we obtain $< t >= \tau$. Ideally if the experiment time goes to infinity this can be applied to any possible value of the lifetime constant. However, in our experiment, we have a observation window which is the time we have our SPAD measuring, and due to this time it is easy to find that the lifetime constant mus be less than four times de time window.

$$< t >= \int_0^{t_w} \frac{dP}{dt} t dt = \tau - e^{-\frac{t_w}{\tau}} (\tau + t_w) \simeq \tau \ \ \forall \tau \in [0, t_w/4] \tag{2}$$

This is one inconvenient of this distribution because the lifetime constant is very restricted for a fixed measuring time. However is interesting because it can converge more quickly than the other method.

$$\epsilon_{\%} = 100 \left| \frac{\tau - < t >}{\tau} \right| = 100 \left| \frac{\tau/\sqrt{N}}{\tau} \right| = \frac{100}{\sqrt{N}} \tag{3}$$

In the next figure (6) the estimated error of the CMM is plotted.

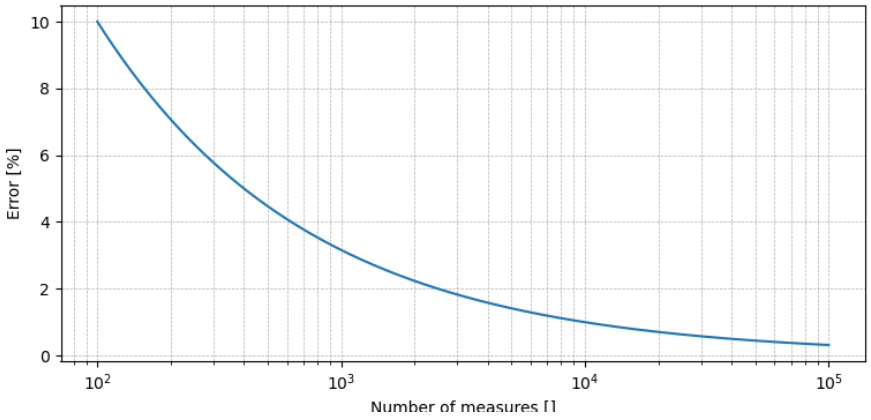



Figure 6: CMM algorithm error as a function of the number of samples

### 4.2.2 Regression Algorithm

The RA performs a linear regression of the data points contained in the bins converted to logarithm. The lifetime constant now is not limited by the expected value integral. So its range can be whatever.

$$\rho\left(t\right) = \frac{dP(t < t, k > 0)}{dt} = \frac{1}{\tau} e^{-\frac{t}{\tau}} \tag{4}$$

And if the logarithm is made we have a linear equation which coefficient can be found by using the method of least squares. (The factor N samples appears when the distribution is normalized to N events)

$$log\left(N\left(t\right)\right) = log\left(\frac{N_o}{\tau} e^{-\frac{t}{\tau}}\right) = log\left(\frac{N_o}{\tau}\right) - \frac{1}{\tau} t \tag{5}$$

Where the slope of the function/line will be $m = -1/\tau$

However the main disadvantage is that we have two sources of error which makes the convergence slower. One comes from the probability distribution seen in the section before, and another one that comes from the linear regression.

To find the error theoretically is more complex due to the different algorithms we use to make the regression. For example, in the logarithm, we make a approximate regression of the points.
In the detail section5.3 we will compare the methods using a python script that executes the calculations and the ModelSim simulations of the RISC running the algorithm.

# 5 Detail Engineering

In this section, a more detailed view of the blocks will be done.

## 5.1 TOP CHIP

### 5.1.1 RISC-V CPU

The RISC-V core chosen for this work is the *SERVRisc* provided by Olof Kindgren. This core stands out for being one of the smallest 32Bit RISC-V (RV32) core and have a prize about it[4].
Due to its tiny size, it is a bit slow, most of instruction cycles, which are one stage operation, are 36 clock long. Then there are the two stage operations such as memory operations or shift operations are 70 clock long. That's because it processes 1 bit for each clock cycle as it name says (serial risc). That's why it makes unique, making it smaller but slower. In the next figure the CPU block diagram is showed.

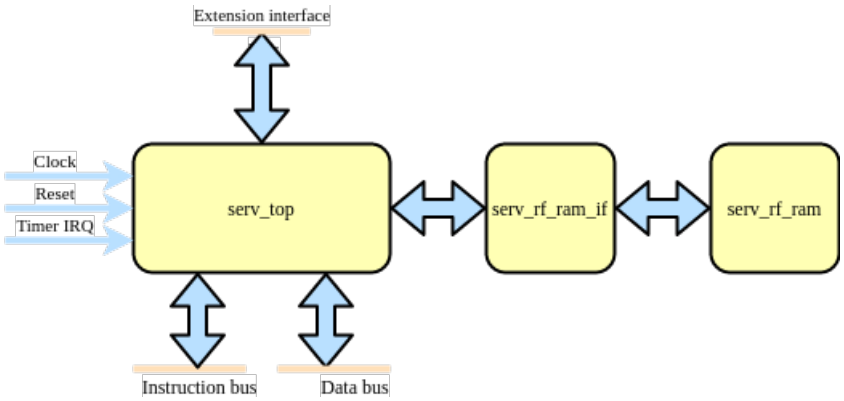



Figure 7: RISC-V CPU block diagram provided by the CPU author[1]

### 5.1.2 SSI or Synchronous Serial Interface

The SSI module, is a slave serial communication module, which works with 8-bit word length and can be able to communicate in the four possible modes. These modes determine when the data is shifted and sampled.

Table 2: SSI functionallity modes

| SSI Mode (CPol, CPha) | Data is shifted out on | Data is sampled on |
|---|---|---|
| **00** | falling SCLK, CSn activates | rising SCLK |
| **01** | rising SCLK | falling SCLK |
| **10** | rising SCLK, CSn activates | falling SCLK |
| **11** | falling SCLK | rising SCLK |

With the SSI slave module, there are two First In First Out memorys (FIFOs), one for transmission and an other one for reception. This FIFOs can generate interrupts which will go to the interrupt module. In the

[4]This prize was given by RISCV org Prize link, Prize link 2

next figure (8) a block diagram of the SSI module is showed (all the wires connections inside the module are not showed). As it is said, the FIFOs can generate different types of interrupts, the SSI interrupts are:

- One or more words received
- Reception FIFO full
- Transmission FIFO empty

Each FIFO is 8 word length and each word is 8 bit length. All the testbench of the FIFOs and the SSI module were done and can be found on the *RV_SoC/RISC-V TOP/FIFObench* and *RV_SoC/RISC-V TOP/SPImoduleBench* in the *github* repository[13].

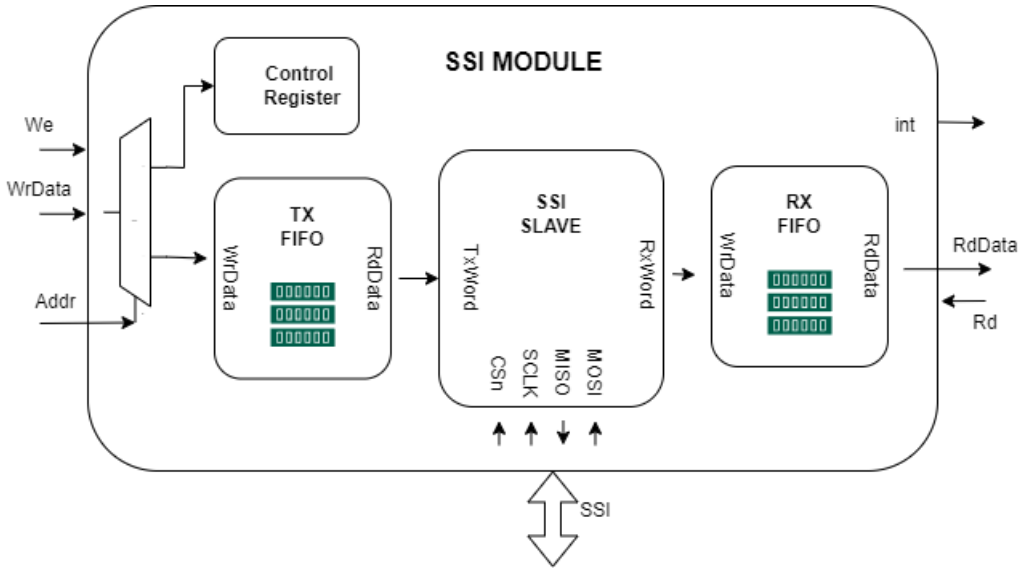



Figure 8: SSI block diagram

### 5.1.3 GPIO

The GPIO module contains 8 pins that can be configured whether as and input or as an output. By default they are defined as input and you can read its digital value and generate negedge interrupts with it. One important thing about the interrupts is that when the interrupt flag is enabled it have to be cleared manually by the processor (put a 0 to the Interrupt Status Register (section 11.1.2). If they are defined as output they can take digital values $(0 - 1)$.

### 5.1.4 Interrupt module

Our RISC-V core only have 1 port for interrupts and our system have different interrupt sources. So a interrupt module have been created in order to join all the possible interrupts to generate a clean interrupt signal. Every interrupt source have to be enabled by accessing the interrupt enable register (section 11.1.3).

One problem found when this *global* interrupt was generated was that is only evaluated when the processor starts a new instruction cycle. In other words, if 1-clock width interrupt starts in the middle of an instruction cycle (minimum 36 clock-width) nothing will happen. So the when a module triggers an interrupt the interrupt module will put a 1 to the *global* interrupt signal until a new instruction cycle begins.

When an interrupt is triggered the firmware will go to a function which will read and clear the interrupt status register in order to know which was the interrupt source.

### 5.1.5 Bootloader Module

The Bootloader or Boot module is responsible for programming the instruction memory when the program needs to be uploaded.

This module is made of a state machine which receive words from the SSI slave and it write them to the instruction memory (IMEM), all this process while is holding the reset for the other modules.

To activate this module a general reset needs to be done, and when the reset push finishes, if the **PROGn** pin has a 0 logic, we will enter to the programming mode by activating the Boot module. On the other hand, if there is a 1 logic, it will be a normal reset maintaining the Boot modulo disabled.

The steps to program the chip correctly are the next ones:

1. Set down the RSTn signal
2. Set down the PROGn signal
3. Set up the RSTn signal. The reset is finished and the Boot module will start working
4. Send by SSI the write instruction byte, in this case 0x02. Now the Boot module will start collecting the instructions of the program
5. Send the program by SSI, LSByte first, MSBit first.
6. When all the program is sent, 0xFFFFFFFF is sent and the bootloader will know that is the end of the program[5].
7. Now once the RISC-V is programmed we can simply restart the chip and/or we can read the memory in order to avoid possible faults.
8. For reading the memory, the instruction to be sent firstly is $0x0x1$ and once is send the memory address to be read needs to be sent by SP¡SI.
9. In the next 32-bit word, the instruction of the address putted before will be sent by the RISC:

### 5.1.6 Timer Module

As it was said the timer module is very simple. There's a timer counter of 32b which will start counting when a register value is written. When it reaches the expected value putted in the register it triggers an interrupt to the interrupt module.

[5] There is no RISC-V ISA which is 0xFFFFFFFF

## 5.2 Firmware

In this subsection a brief explanation of the developed firmware will be done. All the firmware files can be found in the github[13] in the sections named firmware.

In order to program the processor we can divide the process in different parts and files. Firstly we are going to explain the different files and then the process to compile the program.

### 5.2.1 Firmware software files

The firmware files can be divided in that files needed to compile and link the code and the code by itself.

- Files to **make** and **link** the program
  - **Makefile** This file is responsible for calling the compiler (GCC compiler[6]) and taking all the necessary files make the hex file containing the final program in words of 32b. In our case the makefile was taken from different sources such as the SERV github[1] but was modified adding some python scripts in order to have a clean 32b hexadecimal word.

```
CROSS=riscv32-unknown-elf-
CFLAGS = -march=rv32im -Wl,-Bstatic,-T,sections.lds,--strip-debug -
    ffreestanding -nostdlib -mstrict-align -O0


firmware.elf: sections.lds start.s firmware.c
  ${CROSS}gcc -DCLK_FREQ=50000000 ${CFLAGS} -o $@ start.s firmware.c
    RISCV_HW.c



%.hex: %.elf
  ${CROSS}objcopy -O verilog $< $@

%.lst: %.elf
  ${CROSS}objdump -d $< >$@



all_noflash: firmware.elf firmware.hex firmware.lst

run_make_word: firmware.hex
  python3 makeWord.py


all: clear all_noflash run_make_word

clear:
```

[6]The RISC-V compiler toolchain is available in LINUX an can be obtained following the easy steps explained by O.Romera in his github repository, otherwise a shell file (made by O.Romera) can be found in our repository to install the dependences, called *InstallRVenvironment*

```
27  rm -rf *.elf
28  rm -rf *.hex
29  rm -rf *.lst
30  rm -rf *.v
31
```

When this makefile is called, the line which is executed is the **all**. Which first will call the **clear** line which removes the files from the past compilation. Then runs the **all_noflash** line. This instruction calls the RISC-V GCC compiler, in our case the RV32IM (we have multiplication module) and we insert all the code files, in our case, the start.s, the firmware.c and the RISCV_HW.c (library). That will generate different types of files, the most important are .hex file (which contains the code) and the .lst file (which explains the program with RISC-V ISA instructions, very useful to debug the processor). Once we have the hex file it have to be converted to the file we want calling a python script.

- **section.lds** That is the linker file and its very important for the compiler because it indicates how the memory is distributed. The programmer can control how the sections are merged, and at what locations they are placed in memory through a linker script file. In our case the linker file specifies the location and length of the data and instruction memory and defines the different data sections inside the data memory.

- **Code**
  - **start.s** This code file is written in RISC-V ISA assembly and it is the first part of the code to be executed. Mainly it clears all the register file values. So the file clears the 32 registers and then it puts a value to the stack pointer. This was a rather heavy problem in the process because until I set a value it did not work properly. The value to be used has to be the maximum address value of the data memory, because each time a function is called, the stack pointer is reduced by N memory locations according to the function.
  - **firmware.c** This is our C code, usually called main, is the software block that is executed once the start.s is executed. This is the file we need to edit to put our programme in.
  - **RISCV_regs.h** This header file contains information about the hardware and it was made to facilitate the code and make it more understandable. **RISCV_HW.h / RISCV_HW.c** This is a C library made of functions to control the hardware. It can be defined as a driver library. The file contains the functions to enable/disable the interrupt (call assembly instructions to make this), then it have functions to manage the different modules such as the interrupt module, the GPIO module and the SSI module. Finally it contains a function to define the function called when a interrupt is triggered[7] and the interrupt function.

### 5.2.2 Firmware process

The process to load a program to our RISC-V core will be explained. In our case it is made using the SSI of a texas instruments microcontroller (TIVA C). The code can be found in the repository[13] in the file of programTIVA.

1. Have the code in the firmware.c file

[7]The function writes the interrupt function address to the MTVEC register which contains the address of the instruction called when an interrupt is triggered

2. Compile the code using the make file. Command *make all* in the directory
3. Once the program is in machine hex code, execute the python script create_hex_file in order to make the hex code understandable for the code composer (texas instruments).
4. Once you have the Intel HEX file go to the code composer debugger and going to the option of Run->Load->Load program you can select the hex file and put it to the 0x00020000 address. Then the program will read this position of the FLASH and will send it by SSI.
5. Run the TI program in the TIVA C connected to the RISC-V.

Finally, a test was carried out where the RISC-V core was put into an FPGA. Externally, a TIVA C (texas instruments) was used to program the processor in the FPGA. The program used was switching an LED (GPIO) with an interrupt from a timer and switching another LED (GPIO) when receiving an interrupt from an input GPIO.

The test files can be found on the github[13] in the *SIM_DE2_115_RISCV_extPROG* file.

## 5.3 Algorithms introduction of algorithm performance

In the next figure we compare the two methods in a 2D graph. Where in one axis (horizontal) we have the lifetime constant value, in units of percentage of time window we are measuring (each time window is 1 bin of time unit, in our circuit each time interval of the Time to Digital Converter (TDC)). And in the other axis (vertical) we have the number of experiments made, important to see the convergence. That's graphs are made in Python, where firstly a random exponential distribution is generated and distributed in 64 for bins, for the being processed by the algorithms.

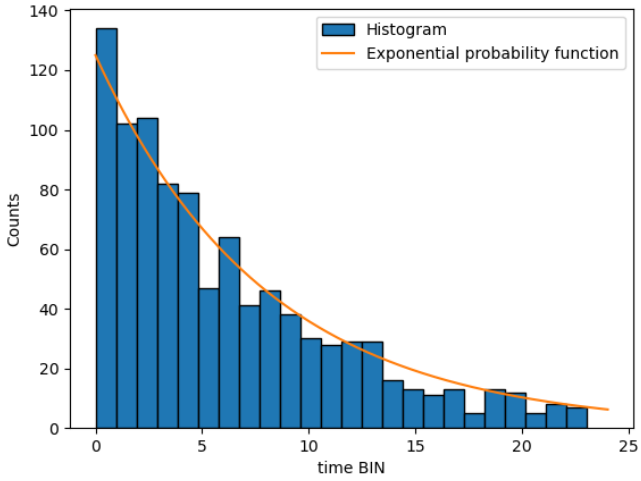
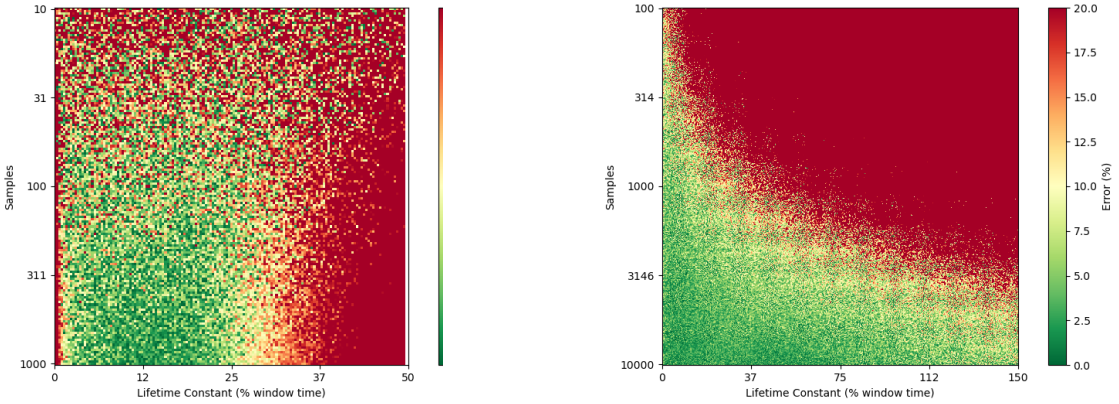



Figure 9: Generated histogram

And if we compute the lifetime constant using the algorithms in a Python program in a PC we get the next results.




(a) CMM Method convergence (b) Regression Method convergence

Figure 10: Convergence of lifetime constant in both methods

We can see that the CMM algorithm converges more fast than the RA algorithm. However, for enough number of samples, the RA algorithm is good for any number of the lifetime constant.

So we are going to implement the Regression Algorithm and the CMM algorithm.

### 5.3.1 Simulation of the algorithms

So first let's going to revise the steps to compute the lifetime algorithm. First of all, we have in a group of registers the counts of the photon hits. This counts make an exponential form which we have to compute the slope magnitude. So the RISC-V will read each one of these registers and compute the logarithm. One problem about our RISC core is that it does not have any floating point unit, so we will need to work with integers and scale factors.

To make the logarithm computation faster what is done first is finding which is the nearest number which logarithm is an integer (in the base of 2). For example if we are computing the *log2* of 5 the nearest number is $2^4$ so 2. Once we have found the nearest number we compute a regression between this number and the next integer number and we find approximately the logarithm. The entire code can be found in the annexes part 11.3.2.
This algorithm has an associated error that depends on how much far are we from the nearest integer logarithm. This error is showed in the next figure.

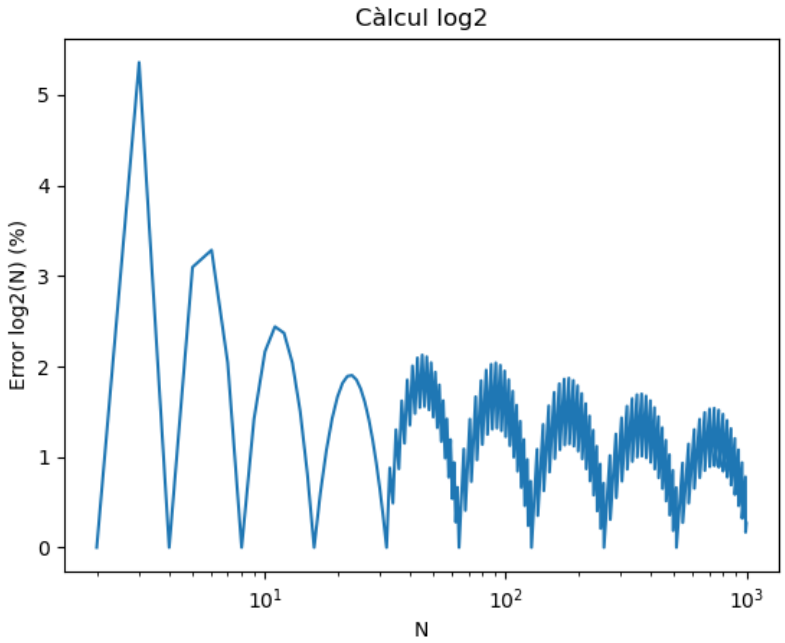



Figure 11: Error of the logarithm algorithm with a scale factor $logF = 4$

We can see a kind of curling for bigger numbers, that if we make the factor bigger will disappear, but otherwise we can have a bit overflow working with bigger numbers.
The associated error can be reduced to $\simeq 1\%$ putting a scale factor of 6 and for small numbers like 1 to 64 can be saved in a look-up table. (64 values codified in 1 byte each, 8 words).

Once we have each logarithm computation we apply the regression formula to obtain the slope. This formula is obtained with least squares method.

$$\tau = \frac{N_{bin} \sum_k klog\,(n_k) - \sum_k k \sum_k log\,(n_k)}{N_{bin} \sum_k k^2 - \left(\sum_k k\right)^2} \tag{6}$$

Where $n_k$ are the photon counts of bin $k$.
The CMM algorithm is more simple where we only have to multiply each bin number by its number of hits.

$$\tau = \frac{1}{N_{samples}} \sum_{i=1}^{N_{bins}} i \cdot n_i \tag{7}$$

Now we are going to simulate the algorithms. The results are obtained from executing the code directly on the RISC-V core in the ModelSim environment. In order to explain better the simulation curse a block diagram is showed.

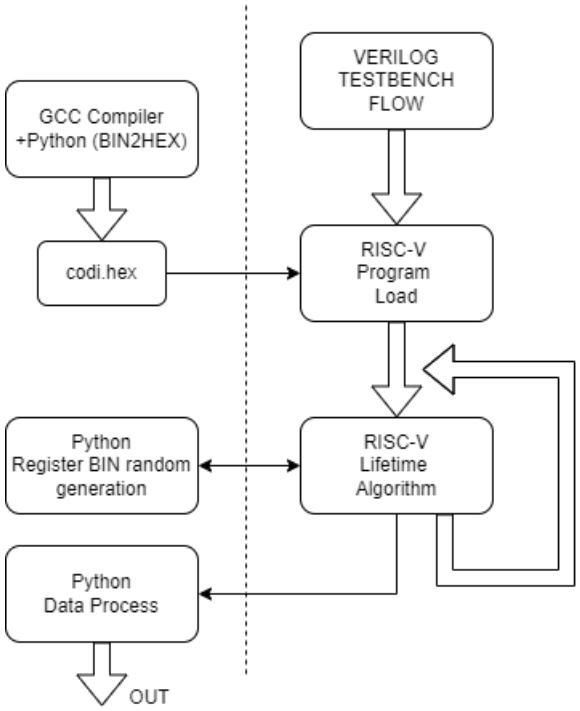


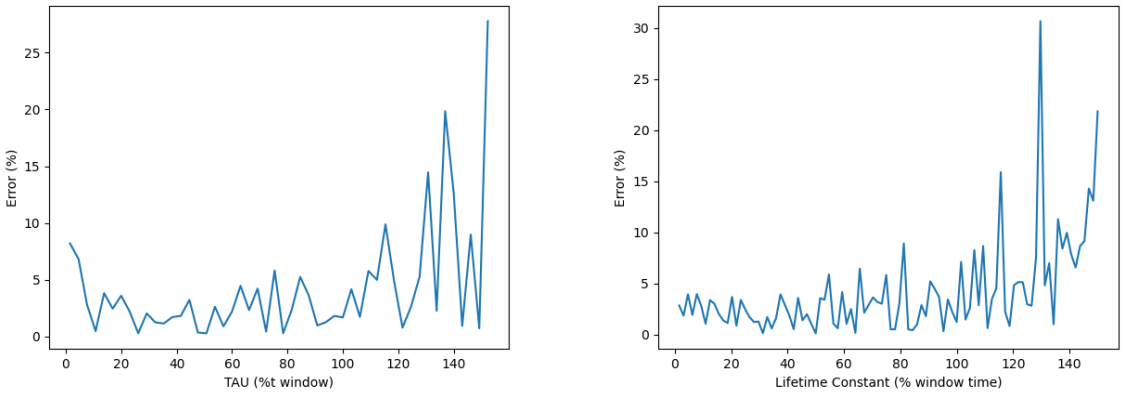

Figure 12: Simulation block diagram

Where while the ModelSim simulation is running it needs the RISC-V code in order to program it by SSI, then it needs a python script to generate the histogram and finally and other one to process the data such as the final lifetime constant and its error and the program time duration.

### 5.3.2 Results of Regression Algorithm (RA)

First, we loaded the program which contained theRA algorithm in order to calculate the lifetime constant.

In the next figure a comparison of results is shown. First, the results of running the C program in the RISC core and in the second figure the results of running a Python program in the PC which computes the lifetime constant. The graphs do not need to be exactly the same because of the fact that when a random distribution is generated not always the results are exactly the same.




(a) ModelSim RISC-V simulation (b) Python PC simulation

Figure 13: Lifetime Constant error using the RA algorithm

We can see that the results are quite similar despite they are obtained from different sources (both hardware and software).

The simulation time was measured to and it can be viewed in the next graph.

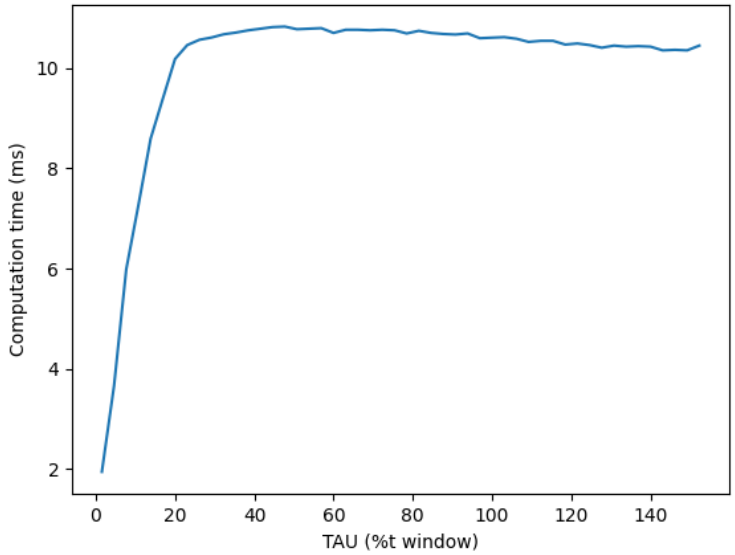


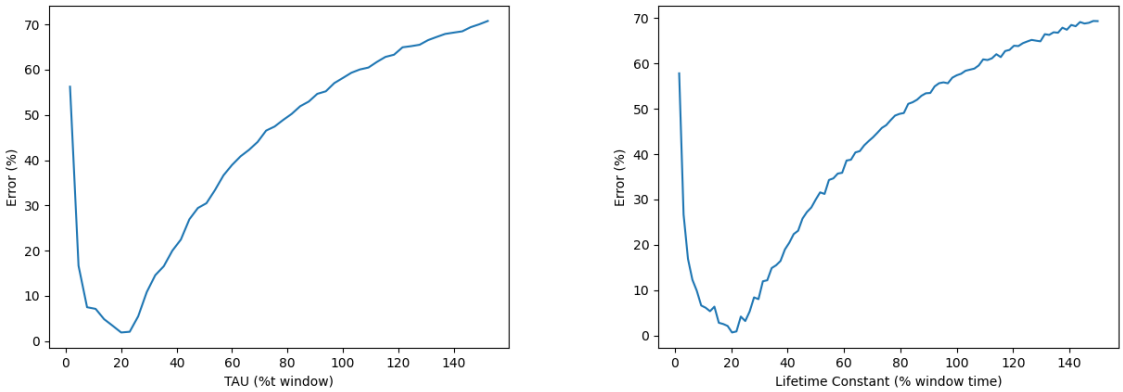

Figure 14: Program duration for RA algorithm when the processor works at $f = 50MHz$

The simulation time is more or less 10ms, which at $f = 50\ MHz$ are $500k$ clock ticks. That's a lot of instructions, maybe if the code is made more optimized (for example in assembly) it can be faster.

### 5.3.3 Results of CMM Algorithm

In the next figure a comparison of results is shown but now using the CMM algorithm. First, the results of running the C program in the RISC core and in the second figure the results of running a Python program

in the PC which computes the lifetime constant. The graphs do not need to be exactly the same because of the fact that when a random distribution is generated not always the results are exactly the same.




(a) ModelSim RISC-V simulation (b) Python PC simulation

Figure 15: Lifetime Constant error using the CMM algorithm

We can see that the results are quite similar too.

The simulation time was measured to and it can be viewed in the next graph.

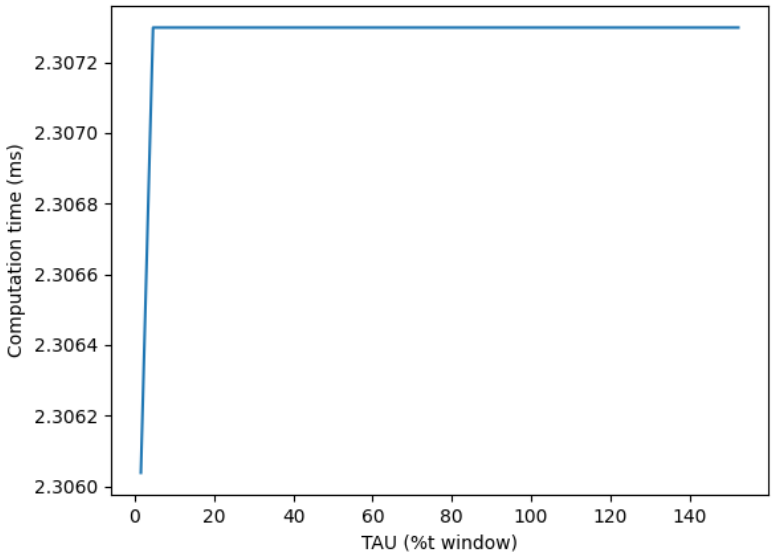



Figure 16: Program duration for RA algorithm when the processor works at $f = 50MHz$

The simulation time is more or less 2.3ms, which at $f = 50\ MHz$ are $115k$ clock ticks. That's a lot of instructions, maybe if the code is made more optimized (for example in assembly) it can be faster as in the first algorithm.

## 5.4 Logic Synthesis

A first approximation for logic Synthesis is made in order to know the most important characteristics of our system.

The technology node chosen is the *XFAB xh018* a 180 nm CMOS technology. And all the design was synthesized in the *Genus* environment. Due to the fact that we have to implement to different memories for instruction and data, different methods are revised. First of them is to directly synthesize the memories as simple registers, which its advantage is mainly the speed.

The other method, more interesting, is to implement memories IP from XFAB which are specifically designed to the characteristics we want. In our case, the most important thing will be the power consumption and then the area/density of the memory. Lastly the velocity as we are not willing to work at very highs speeds.

Once we have done the first synthesis of the design we obtained the next results.

Table 3: Synthesis Results table

| **Feature** | **Result** |
|---|---|
| Cell Area ($\mu m^2$) | 1572307 |
| Cell Area ($\sqrt{cm^2}$) | 1.254 |
| Instances (without IP) | 15278 |
| Power (1st approach)($mW$) | 18.18 |

In terms of power consumption, it is important to add that this is an estimated value of power is obtained considering all the combinational and sequential logic working,. However, not always all the registers will be operating. To have a better number, the simulation files should be provided to get a better estimation. We can take the current value as the maximum consumption limit.

We have used a IP block. Especially a 128x16 bits memory, 16 blocks for the instruction memory (1kiWords, 4 kiBytes), and 8 blocks for the data memory (512 Words or 2 kiBytes).

## 5.5 Power Analysis

The Power Analysis is very important since one of the project's objectives is to have a decrease of the power due to the reduction of the transmission data.

The power is divided in different parts. First of all we have the power consumption of the logic inside the RISC-V microprocessor. The numbers are obtained from the synthesis tools that use the technology files and the design to estimate the power.

Then we have the power due to the data storage. This ones can be simple sets of registers synthesized by the tool, or specific memory IP's. We will discuss which one is best based on size and consumption. When these two power sources are combined, this results in a power consumption of about $P = 18.18\ mW$.

Finally we have the transmission power, which takes into account the waste of energy to the transmission data to the final receiver. If we put the processor, only 8 bits are needed to define the time constant. However, if the RISC-V is not used we will need: If we make 10k samples in 64 bins, each bin 10-bit length, 640 bits to be transmitted.

That is to say, we went from 8 bits per pixel to 640 bits per pixel. That is 80 times more data and if the data has to be sent telematically it can reduce the consumption a lot. As well as storing the time constant in the RISC-V you can avoid continuously sending data.

# 6 Regulations and Legal Aspects

Because the RISC-V ISA is open source and royalty-free, there is no licence to worry about. Also our RISC-V core is open source and the other modules are developed by us or are open source too.

On the other hand, as this project does not cover the physical design of the product there is no need to worry about consumer/electronic device standards.

# 7 Technical Viability

In this section the technical viability of the project will be discussed.

## 7.1 DAFO analysis

SWOT stands for **S**trengths, **W**eaknesses, **O**pportunities, and **T**hreats, and so a SWOT analysis is a technique for assessing these four aspects of your business/project. The SWOT analysis of the project is presented below.

- **Strengths** One of the main strengths of the project is its innovation and advantages in terms of energy consumption and the ability to process the information directly in the sensor itself.
- **Weaknesses** The main weakness is that it takes a lot of time, money and testing to get the final chip design.
- **Opportunities** One of the main strengths is the innovation of the project in the market and the expertise of the faculty department in the field of SPADs.
- **Threats** As a microelectronics project, the main competitors are usually multi-million dollar companies with a lot of resources. It is therefore necessary to protect and/or sell the idea very well.

# 8 Economic Viability

We can split the costs of this project in different parts. First we have the labour of minimum one engineer. Then we have the software, all mainly free expect Cadence Genus (synthesis), which is quite expensive, but as the university has licences we will assume that it is free of charge. We are not going to tell the whole cost of making the chip and testing it as the project does not cover everything, we are only going to mention how much it would cost to send the chip to manufacture.
Looking at europractice web we can see that in our technology, the $mm^2$ is 2025€ so if our RISC-V area is $1.58\ mm^2$ the price is 3185€. However the minimum area is $10\ mm^2$ so the die will cost 20250€.
Then if we consider the labour of an electronic engineer, considering that the project have a duration of 13 weeks (9.3), and the salary per month of an engineer is about 3000€ we can expect a cost of $3000€ \cdot 13/4 = 9750€$.

Then if we consider the IT material costs we have:

- PC + screen + electricity : 1000€
- Server : $20000 \frac{€}{5\ years\ \ 10px} \simeq 400€$

So the main cost of the project without the fabrication stage / test is around 9750€ and if we take into account the die cost is $9750 + 20250 = 30k€$ (All this without taking into account the possibility of sending the chip to be manufactured more than once, as well as the need + amortisation of test equipment).

# 9 Schedule Execution

In the next section we are going to explain the schedule execution by describing the **W**ork **B**reakdown **S**tructure (WBS).

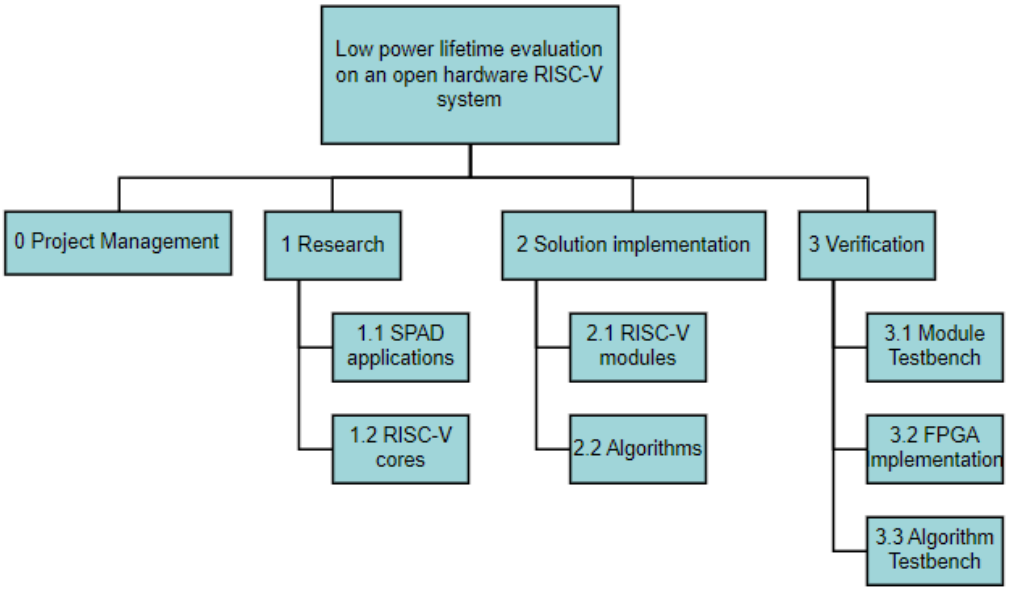



Figure 17: **W**ork **B**reakdown **S**tructure block diagram

## 9.1 WBS Dictionary (Work Packages)

The different parts of the WBS are:

- **Project Management** Plan, organize and manage the resources to complete the project.
- **Research** First approach to the project, SPAD applications are reviewed in order to define the objectives for then design the hardware/software. Also is studied the RISC-V ISA and cores in order to understand how a RISC-V CPU works.
- **Solution Implementation** Design the hardware/firmware modules for processing the data. Create the algorithms for finding the lifetime constant.
- **Test** Make the test to verify the hardware modules. Test the microprocessor in a FPGA. Finally run the algorithm in the hardware modules.

## 9.2 Precedence Analysis

Table 4: Precedence analysis

| WBS Reference | Activity | Precedence | Duration (weeks) |
|---|---|---|---|
| 1.1 SPAD Applications | A | - | 1 |
| 1.2 RISC-V Cores | B | - | 1 |
| 2.1 RISC-V Modules | C | B | 8 |
| 2.2 Algorithms | D | A | 4 |
| 3.1 Module Testbench | E | C | 4 |
| 3.2 FPGA Implementation | F | C | 2 |
| 3.3 Algorithm Testbench | G | C,D | 4 |

## 9.3 GANTT and CPM charts

The Gantt chart is a type of bar chart that illustrates a project schedule. In the next picture (18), the Gantt chart is shown.

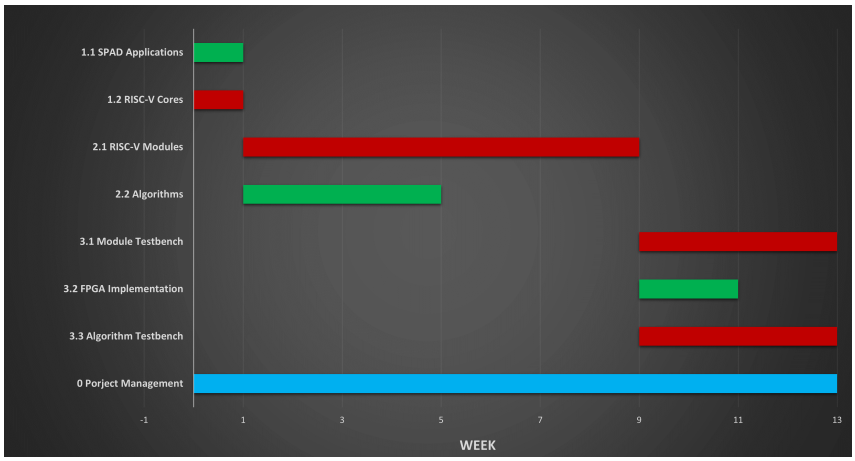



Figure 18: Gantt chart

The CPM diagram takes the activities and shows the schedule of them in time as we can see in the next figure (19).

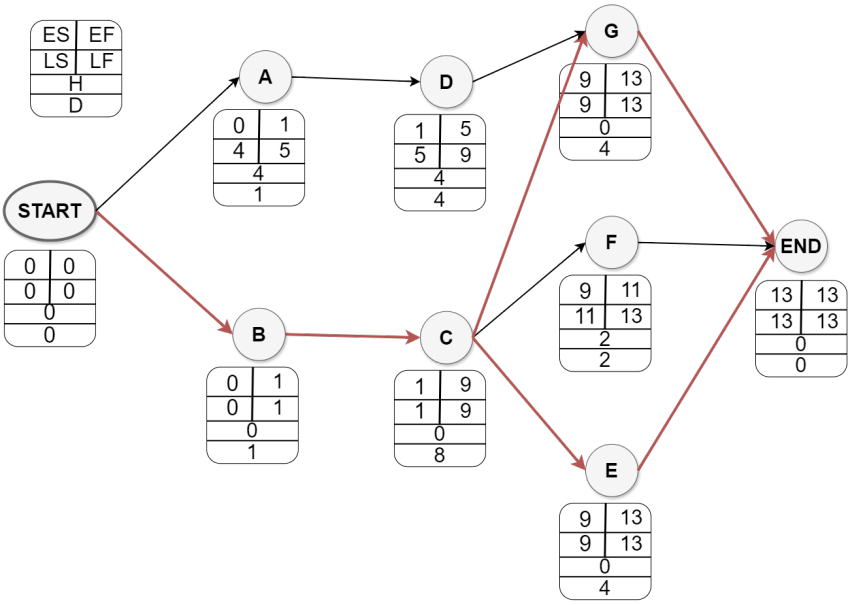



Figure 19: CPM chart

# 10 Conclusions

As already mentioned, putting a processor next to the sensor can have many advantages. And by making it reprogrammable, different processing algorithms can be tested and optimized.
In this work it has been shown that different algorithms can be applied to a RISC-V core to process data for certain applications. Although being so small in size makes it a slow processor as seen in the results section (ref) makes us rethink other RISC-V cores that have shorter instruction cycles (the current one has between 36-72 clocks and if it can be reduced more than 10 times).

As far as the algorithms are concerned, the two proposed methods have been tested and each of their advantages has been demonstrated. Looking at the results we can see that the CMM method (4.2.1) thanks to its fast convergence can be more interesting if the task requires speed, however if we want more precision we can opt for the regression method (4.2.2).

On the other hand, it has been possible to verify that the hardware blocks work correctly, especially the programming block (5.1.5) and the SSI communication block (5.1.2). It has also been interesting to make a first approach to the chip synthesis (logic synthesis), as it has been possible to see the area and power consumption.

Finally to conclude, this project may also be interesting to extrapolate to other Edge Computing applications. In other words, in this project we have worked with SPADs, however, the RISC-V core has been made quite generic and could be applied as an edge device for other applications.

However, this document ends here up to the RTL level of the design mainly due to time constraints, we hope to continue to work on it and finally put it together (in hardware) with the SPADs modules (which still need to be modified) and start to implement it on a chip.

# 11 Annexes

## 11.1 RISC-V core USER GUIDE

In this sub-document we will revise the RISC-V TOP capabilities and functionalities showing its register map.

### 11.1.1 SSI Module

The SSI have two main registers, each one 16-bit length. The first register is for reading/writing data to the FIFOs and the second register configures the slave module and the FIFOs of transmit and receive. The SSI direction offset is **0x20000000**.

Table 5: SSI Data Register (**+0x00**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 7:0 | SSIDR | R/W | X | Data to write/read to the SSI interface |

Table 6: SSI Configuration Register (**+0x04**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 15 | TXIM | R/W | 0 | Interrupt Mask for TX FIFO empty event |
| 14 | RXIM | R/W | 0 | Interrupt Mask for RX FIFO full event |
| 13 | RTIM | R/W | 0 | Interrupt Mask for any word received |
| 12:8 | BS | R/W | 0x08 | SSI word size in bits |
| 7:6 | CPP | R/W | 0 | SSI CPol, CPha mode |
| 5 | SFF | W | 0 | If 1, the TX and RX FIFO will be rst. |
| 4 | RXFF | R | 0 | RX FIFO full status |
| 3 | RXFE | R | 1 | RX FIFO empty status |
| 2 | TXFF | R | 0 | TX FIFO full status |
| 1 | TXFE | R | 1 | TX FIFO empty status |
| 0 | BSY | R | 0 | SPI busy bit (i_CSn value) |

### 11.1.2 GPIO Module

The GPIO module has 4 registers, each one to control different aspects of its functionality.
The GPIO direction offset is **0x30000000**.

First of all, we have the register that saves the input data of the GPIO, if it is defined as input we are reading the input value, and if it is defined as output we actually obtain the value we have given to the GPIO.

Table 7: GPIO Read Register (**+0x00**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 7:0 | getGPIO | R | X | GPIO input value |

Then we have the configuration register that if the pin is declared as input, will enable the interrupt of the pin and if it is declared as output will set its output value.

Table 8: GPIO Out Register (**+0x04**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 7:0 | setGPIO | R/W | 0 | If declared as input, interrupt enable. If declared as output, output value. |

The next configuration register, sets the pin direction.

Table 9: GPIO Direction Register (**+0x08**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 7:0 | dirGPIO | R/W | 0 | If 0, Input<br>If 1, Output |

Finally there is the interrupt status register, that contains which was the register interrupt source. When an GPIO interrupt triggers this register have to be set to zero manually.

Table 10: GPIO Interrupt Status (**+0x0C**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 7:0 | isGPIO | R/W | 0 | Interrupt Flag |

### 11.1.3 Interrupt Module

Because of the fact that the RISC-V Central Process Unit (CPU) only supports one interrupt source (timer interrupt) we need a module that manages all the different interrupt sources. This module support eight different interrupt sources.
The Interrupt direction offset is **0x40000000**.
There are two different registers, one for enabling the interrupts and another one to view the interrupt status.

Table 11: Interrupt Status (**+0x00**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 7 | IS P7 | R/W | 0 | Port 7 interrupt status |
| 6 | IS P6 | R/W | 0 | Port 6 interrupt status |
| 5 | IS P5 | R/W | 0 | Port 5 interrupt status |
| 4 | IS P4 | R/W | 0 | Port 4 interrupt status |
| 3 | IS P3 | R/W | 0 | Port 3 interrupt status |
| 2 | IS Timer | R/W | 0 | Timer interrupt status |
| 1 | IS GPIO | R/W | 0 | GPIO interrupt status |
| 0 | IS SSI | R/W | 0 | SSI interrupt status |

And then we have the interrupt enable register, which each bit enables a different interrupt source.

Table 12: Interrupt Enable (**+0x04**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 7 | IE P7 | R/W | 0 | Port 7 interrupt enable |
| 6 | IE P6 | R/W | 0 | Port 6 interrupt enable |
| 5 | IE P5 | R/W | 0 | Port 5 interrupt enable |
| 4 | IE P4 | R/W | 0 | Port 4 interrupt enable |
| 3 | IE P3 | R/W | 0 | Port 3 interrupt enable |
| 2 | IE Timer | R/W | 0 | Timer interrupt enable |
| 1 | IE GPIO | R/W | 0 | GPIO interrupt enable |
| 0 | IE SSI | R/W | 0 | SSI interrupt enable |

### 11.1.4 Timer Module

The Timer direction offset is **0x50000000**. The timer is a counter of 32 bits that works with the system clock. It have three main registers but there are only 2 writable/readable directions. In order to write/read the timer enable this will be in the LSB of the timer count register.

Table 13: Timer Control Register (**+0x00**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 31:1 | TCount[31:1] | R/W | 0 | 31 MSB of timer counter |
| 0 | TEN | R/W | 0 | Timer enable |

In the following register is the value at which, when the timer counter reaches it, an interrupt is generated and the counter is cleared.

Table 14: Timer End Register (**+0x04**)

| Bit/Field | Name | Type | Reset | Description |
|---|---|---|---|---|
| 31:0 | TREG | R/W | 0 | Timer Limit Value |

## 11.2 Code

## 11.3 Firmware for the RISC-V core

The next code shows the main firmware functions to control the RISC-V and its modules commented. This code is completely developed by me and it is based on the RISC-V ISA.

```
#include "RISCV_HW.h"

extern void SSI_isr(void); // ISR functions
extern void GPIO_isr(void);
extern void Timer_isr(void);
extern void INT3isr(void);
//extern void INT4isr(void)...

void volatile enable_mie(void){
  __asm__ volatile("li t1, 0x00000008"); // Put 0x08 value to t1
  __asm__ volatile(".word 0x00031073"); //Put t1 to MIE crs register
}

void enable_timer_interrupts(bool estat){
  if(estat){
    __asm__ volatile("li t1, 0x00000080"); //if enable we put 0x80 to t1
  }else{
    __asm__ volatile("li t1, 0x00000000"); //if disable we put 0x00 to t1
  }
  __asm__ volatile(".word 0x00431073"); //write t1 to MTIE csr reg
}

void intRegister(void (*func)(void)){  //function called when timer interrupt comes

  uint32_t value = (uint32_t)((uintptr_t)func);

  __asm__ volatile ("mv t1, %0"
                    : // output: none //
                    : "r" (value) // input : from register //
                    : "t1");
  __asm__ volatile(".word 0x00531073"); //Write t1 to MTVEC
}

void INTENABLE(uint8_t source){
  *(INTM+IE)=*(INTM+IE) | source; //module int enable
```

```
}
void INTDISABLE(uint8_t source){
  *(INTM+IE)=*(INTM+IE) & (~source); //module int disable
}


//----- GPIO functions------------//
void GPIO_Input(uint8_t GPIOpin){
  *(GPIO+dirGPIO) = (*(GPIO+dirGPIO))&(~GPIOpin);
}
void GPIO_Output(uint8_t GPIOpin){
  *(GPIO+dirGPIO) = (*(GPIO+dirGPIO))|(GPIOpin);
}
void GPIO_write(uint8_t GPIOpin, uint8_t GPIO_state){
  *(GPIO+setGPIO) = ((*(GPIO+setGPIO))&(~GPIOpin))|(GPIO_state&GPIOpin);
}
uint8_t GPIO_read(void){
  return (*(GPIO+getGPIO));
}
void GPIO_IntEnable(uint8_t GPIOpin){
  //ha de ser input
  *(GPIO+setGPIO) = ((*(GPIO+setGPIO))&(~GPIOpin))|(GPIOpin);
}

void irq_entry(void)  {
    uint8_t IntVec;

  IntVec = *(INTM);
  *(INTM) = 0; //clean interrupt flag


  if(IntVec&int_SSI){ //SSI interrupt
    SSI_isr();
  }
  if(IntVec&int_GPIO){ //GPIO interrupt
    GPIO_isr();
  }
  if(IntVec&int_Timer){ //timer interrupt
    Timer_isr();
  }
  if(IntVec&BIT3){ //int3 interrupt...
    INT3isr();
  }

}

bool SSI_PutData(uint8_t data){
  uint16_t reg16 = *(SSI+1);
  if(reg16&BIT2){ //if its full
    return false;
  } else{
    *(SSI) = data;
    return false;
  }
}

bool SSI_ReadData(uint8_t* data){
  uint16_t reg16 = *(SSI+1);
  if(reg16&BIT3){ //if its empty
    return false;
  } else{
```

```
97      *data = *SSI;
98      return false;
99    }
100 }
```

### 11.3.1 Regression Algorithm

This algorithm returns de lifetime constant with a scale factor

$$\tau = \tau_{RAW} \frac{log_2(e)}{2^{resTAU}}$$

```
1  #define logF 4       //factor of 16 f.e.
2  #define resTAU 4      //factor
3
4  void calcTAU(volatile uint32_t* regs, uint8_t nRegs, uint32_t* tauRAW){
5    uint32_t sumxsumy =0;
6    uint32_t sumxy = 0;
7    uint8_t nbins;
8    uint16_t dada;
9    uint32_t y;
10   for (nbins=0;nbins<nRegs;nbins++){
11     dada = *(regs+nbins);
12     if(dada){
13       y = log2n((uint32_t)dada);
14       sumxsumy += y;
15       sumxy += y*nbins;
16     } else{ //hem arribat al maxim (primer 0)
17       nbins++;
18       break;
19     }
20   }
21   uint32_t sumx = nbins * (nbins-1);
22   sumx >>= 1;
23     sumxsumy *= sumx;
24   uint32_t Nsumx2 = sumx * (-1+(nbins<<1));
25   Nsumx2 = Nsumx2 / 3;
26   Nsumx2 *= nbins;
27
28   Nsumx2 = (Nsumx2 - sumx*sumx);
29   Nsumx2 <<= (logF+resTAU);
30   sumxsumy = sumxsumy - ( nbins * sumxy );//sumx * sumy
31
32   if(sumxsumy){
33     *tauRAW = Nsumx2 / sumxsumy;
34   } else{
35     //error divisio
36     *tauRAW = 0xFFFFFFFF; //infinit
37   }
38 }
```

### 11.3.2 *log2* algorithm

The following code computes the base 2 logarithm of an integer with a scale factor ($2^{logF}$) where $logF$ is a defined constant.

```
1  #define logF 4       //factor of 16 f.e.
2
```

```
uint32_t log2n(uint32_t x){
  if(x<2){
    return 0;
  } else{
    uint8_t i=0;
    while(x>(1<<i)){
      i++;
    }
    i--;
    uint32_t log = x;
    if(i>logF){
      log >>=(i-logF);
    } else{
      log <<=(logF-i); //2**logF factor
    }
    log+= (i-1)<<logF;
    return log;
  }
}
```

# References


[1] O. Kindgren, "Serv: Bit-serial risc-v core," https://github.com/olofk/serv, 2022.

[2] A. Rochas *et al.*, *Review of Scientific Instruments*, vol. 74, no. 7, pp. 3263–3270, 2003.

[3] E. Charbon, "Spad based image sensors," in *2014 IEEE International Electron Devices Meeting*, 2014, pp. 10.2.1–10.2.4. [Online]. Available: https://ieeexplore-ieee-org.sire.ub.edu/document/7047022

[4] H. Yue, Y. Xu, Y. Huang, and X. Xie, "Fully integrated high density spad array detector," in *2014 12th IEEE International Conference on Solid-State and Integrated Circuit Technology (ICSICT)*, 2014, pp. 1–3. [Online]. Available: https://ieeexplore-ieee-org.sire.ub.edu/document/7021592

[5] Fahim-Ul-Hassan and G. Cook, "Pet/ct in oncology," *Clinical Medicine (London)*, vol. 12, no. 4, pp. 368–372, Aug 2012. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4952129/

[6] V. Zickus, M. L. Wu, K. Morimoto *et al.*, "Fluorescence lifetime imaging with a megapixel spad camera and neural network lifetime estimation," *Scientific Reports Nature*, vol. 10, p. 20986, 2020. [Online]. Available: https://www.nature.com/articles/s41598-020-77737-0

[7] V. Moro, S. Moreno, O. Alonso, A. Vilà, J. D. Prades, and A. Diéguez, "Using time-correlated single-photon counting technique on spad sensors to enhance acquisition time and dynamic range," *Proceedings of SPIE*, 2022, institutional repositories. [Online]. Available: https://www.ub.edu/in2ub/ca/publicaci-peridica/using-time-correlated-single-photon-counting-technique-on-spad-sensors-to-enhance-acquisition-time-and-dynan

[8] M. S. Ara Shawkat, S. Hasan, and N. McFarlane, "Single photon detectors for quantum computing," in *2023 IEEE 16th Dallas Circuits and Systems Conference (DCAS)*, 2023, pp. 1–4. [Online]. Available: https://ieeexplore-ieee-org.sire.ub.edu/document/10130206

[9] QYResearch, *Global Single Photon Avalanche Diode (SPAD) Module Market Research Report 2023*. QYResearch, November 2023, code: QYRE-Auto-32A14056, Pages: 89. [Online]. Available: https://reports.valuates.com/market-reports/QYRE-Auto-32A14056/global-single-photon-avalanche-diode-spad-module

[10] A. Diéguez, J. Canals, N. Franch, J. Diéguez, O. Alonso, and A. Vilà, "A compact analog histogramming spad-based cmos chip for time-resolved fluorescence," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 2, pp. 343–351, 2019. [Online]. Available: https://ieeexplore-ieee-org.sire.ub.edu/document/8611169

[11] J. Canals, N. Franch, A. Dieguez, and UB, "Overcoming the limits of diffraction with superresolution lighting on a chip," Project funded by the European Union Horizon 2020 Programme, 2020. [Online]. Available: https://cordis.europa.eu/project/id/737089

[12] C. X. Wolf, "Picorv32 - a size-optimized risc-v cpu," https://github.com/YosysHQ/picorv32, 2015.

[13] G. Prenafeta, "Risc-v top repository," https://github.com/guillemmmm/RV_SoC, 2024.

[14] H. Xue, B. Huang, M. Qin, H. Zhou, and H. Yang, "Edge computing for internet of things: A survey," in *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, 2020, pp. 755–760. [Online]. Available: https://ieeexplore.ieee.org/document/9291551