Caiet de practică

Student: **Rădoni Tudor-Călin**

Facultatea de **Automatică și Calculatoare**

Specializarea **Automatică**

Grupa: **30332, engleză**

Firma la care s-a efectuat practica: **Analog Devices Inc.**

Perioada în care s-a efectuat practica: **17/07/2023** – **25/08/2023**

Numărul total de ore de practică: **240**

Numele tutorelui din partea firmei: **Adrian Suciu**

Summary

Within our project, we worked to develop a **digital accelerometer** along with the corresponding **software applications**. This device has the capability to measure acceleration and interpret these measurements effectively. To achieve this, we went through several important stages. We began with designing the **electrical schematic** and **layout**. Then, we developed **the driver** that enables communication with the accelerometer, as well as the **software** for interpreting the measured data. In the following sections, I will detail each of these stages and how they contribute to the final operation of the digital accelerometer and its associated applications.

Drawing the schematic with components required for the PCB

This initial stage involves designing the schematic, which is the electronic diagram of all the components needed for the accelerometer. This includes active components (such as the accelerometer chip), passive components (resistors, capacitors, etc.), and any interfaces with other devices. The schematic serves as a guide for correctly connecting the components in the future PCB.

Tools used: LTSpice and KiCad.

PCB Layout

The layout process involves physically placing the components on the printed circuit board (PCB) and routing the electrical connections between them. A well-optimized layout ensures minimizing interferences and critical distances between components, allowing optimal electrical and mechanical performance of the device. More details will be available in the detailed description, where the necessary specifications are described.

Tool used: KiCad.

HDL Part (Hardware Description Language)

In this stage, we develop the HDL code (using Verilog) that describes the functionality of the accelerometer chip at the hardware level. This code is essential for configuring and controlling the chip as desired and for interacting with the rest of the components. Additionally, in this module, we created a PWM generator used to control LEDs on the Zybo board.

Tools used: Vivado, Vitis, TerosHDL, and Visual Studio Code.

Linux Driver

Developing a driver for the Linux operating system enables proper and efficient communication between the accelerometer and the operating system. This driver is in line with upstream preferences, adhering to the standards and requirements of the Linux community. This encourages the acceptance and inclusion of the driver in official Linux distributions.

Tools used: VirtualBox and VMWare, Visual Studio Code, and Qemu.

Development of Applications for the Accelerometer:

Here, applications were developed to make use of the data provided by the accelerometer. These applications include reading data from the accelerometer and interpreting them in appropriate units (G, volts and raw value). Some examples of applications we developed include continuous reading of accelerometer data in tabular form in all the aforementioned units, a digital level with a graphical interface and calibration support.

Tools used: Visual Studio Code and CMake.

Overall, these activities led to the development of a functional digital accelerometer and a set of applications.

Detailed Description of the Activities

Following is a detailed description of the activites. It was all writted in **Markdown** and I used *Visual Studio Code*, along with *Obsidian* to write, edit, export and beautify everything.

Notes

first-week

First week - Electronics

Day 1 - Software installation and basic circuits

The day started with installing loooots of software, amongst which were *Scopy* and *LTSpice*.

Next task? **Do some electronics labs!** They were pretty cool, as we were given ADALM2000 boards and we had to do some basic circuits with them. We also had to use a breadboard and some resistors, capacitors and LEDs.

We had to do some basic circuits, like a voltage divider, a voltage follower, a low pass filter, a high pass filter, a band pass filter and a band stop filter.

Day 2 - More circuits and some more serios things

Today, we had to learn some configurations for using Op-Amps and made a buffer. This helped us build a reference output voltage for the next circuit, which was a voltage regulator. Nothing very much to say about this, as it was pretty easy.

Day 3 - LDO and testing

The task of the day: build a low drop-out voltage regulator. This was a bit more complicated, as we had to use a MOSFET and a BJT. We also had to use a potentiometer to adjust the output voltage.

After most of us were done, we were given a proper LDO encapsulated in a single package.

Day 4 - Accelerometer to Converter interface, LTSpice

Faced with the problem that the ADC takes a maximum input voltage of 3V, we had to build a circuit that would take the output of the accelerometer and convert it to a voltage that the ADC can read. We used a voltage divider and a buffer to do this.

Furthermore, we tested everything in an LTSpice simulation and it worked! Now, on the breadboard! This is the final result, which was tested with Scopy.

> ADXL327

ADXL327 Tests

We have done some measurements to make sure that everything is wired correctly. This also helped us decide what is the middle value for each axis (even though they are all the same). We didn't use the values from the datasheet because they used a 3V power supply and we used 3.3V, so the values are a tiny bit different.

Measurements

X Axis (board pin 12)

```
   min: 1.165
offset: 1.611
   max: 2.058
```

Y Axis (board pin 10)

```
   min: 1.16
offset: 1.66
   max: 2.088
```

Z Axis (board pin 8)

```
   min: 1.159
offset: 1.839
   max: 2.078
```

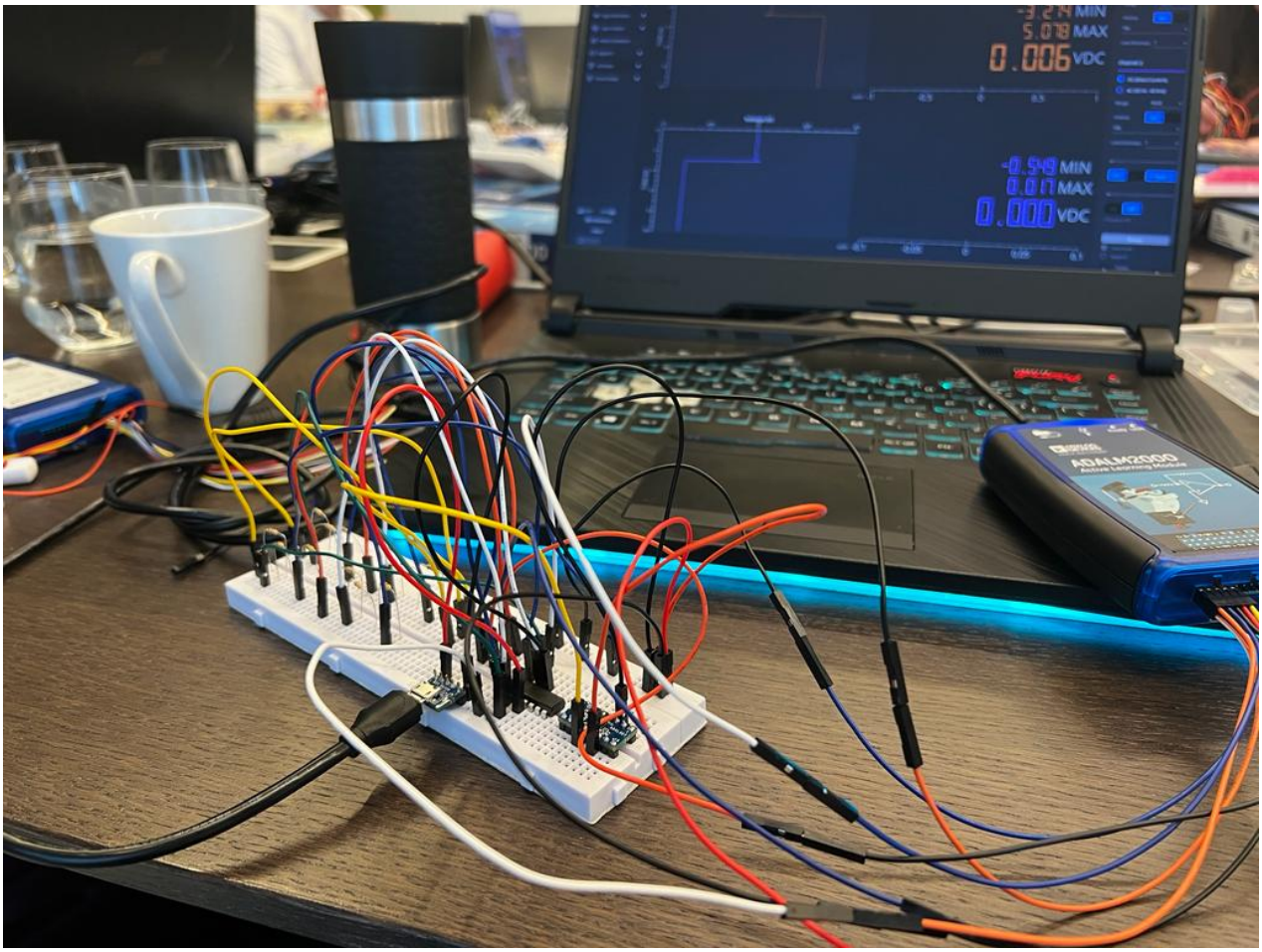
Normalized Values

All axes will be normalized to the same range, therefore:

```
   min: 1.1
offset: 1.6
   max: 2.1
```

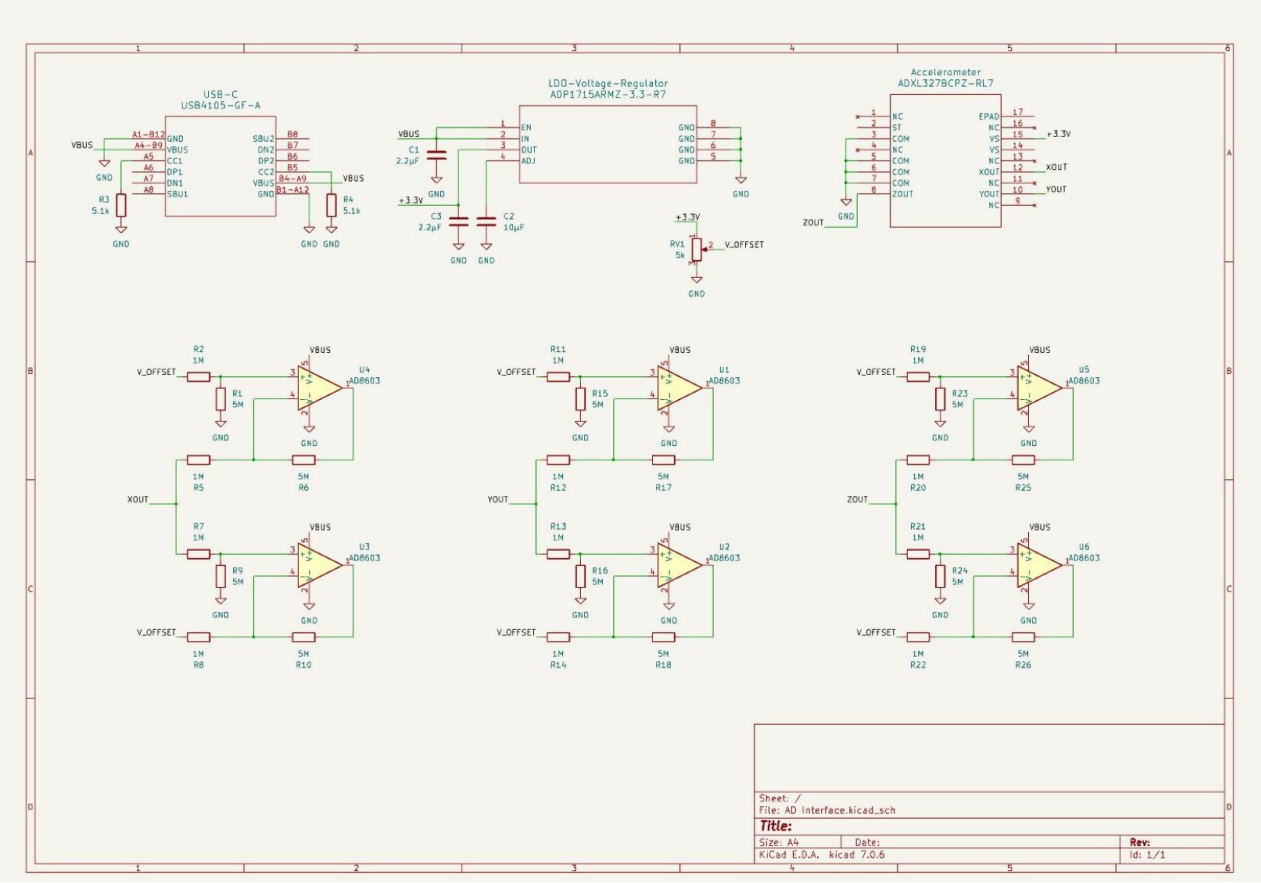
Conclusion

This way, the middle value is `1.6` and the range is `0.5`. Using **op-amps**, we will split the range into 2 parts, one for the **negative** and one for the **positive** values. Moreover, we will use a gain of `5`, so the range will be `0 - 2.5V`. Great! The values are in the range of the ADC, so each of the 6 channels will be able to read the values with a **greater precision** than if we used only 3 channels.



Day 5 - KiCad

Now, we must build the schematic. For this, we got familiar with KiCad, UltraLibrarian and we got to build the schematic for the accelerometer to converter interface.



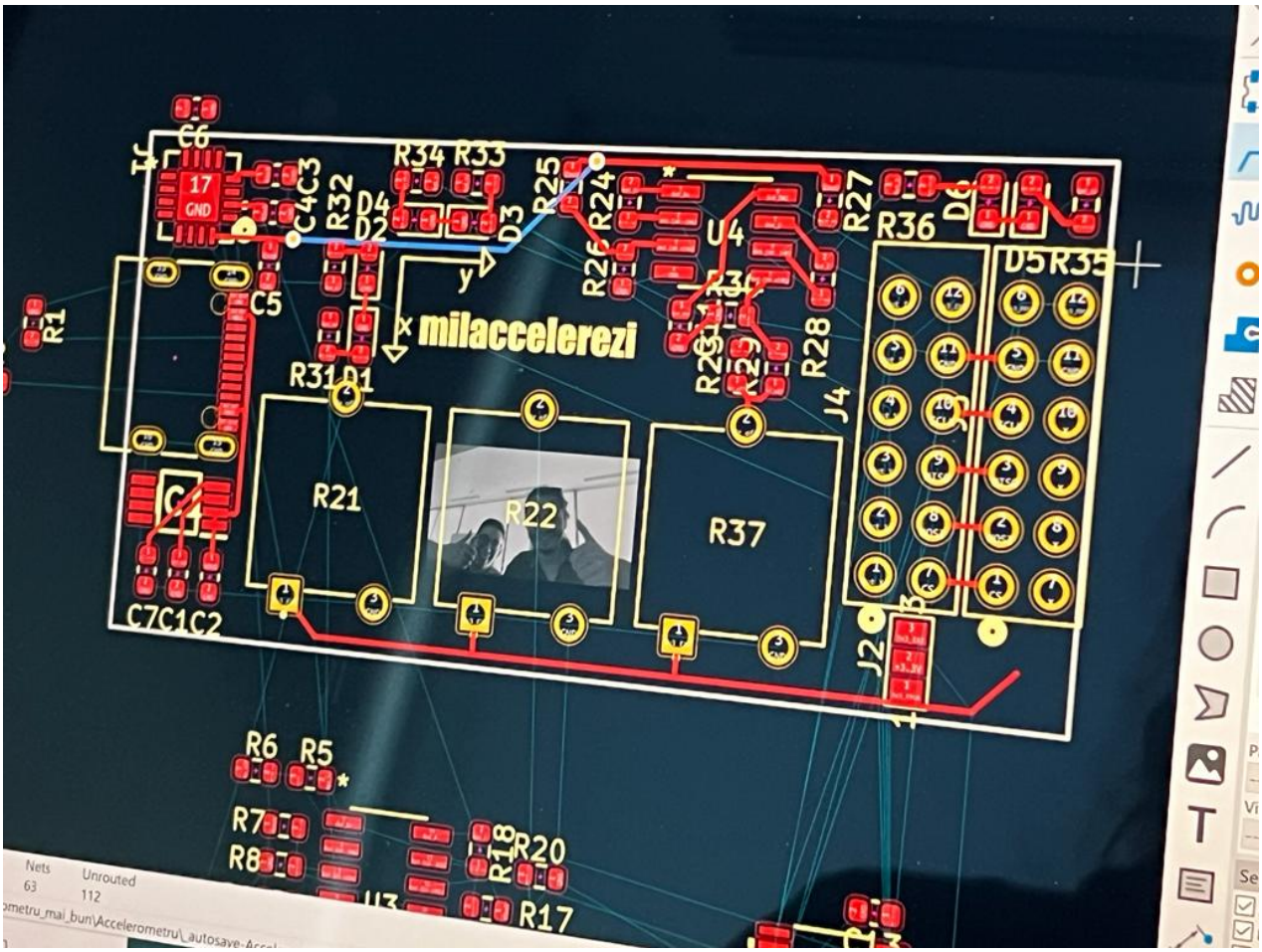
Day 6 - PCB Layout

Having the schematic all ready and verified by the mentors, we started building the PCB layout. This was a bit more complicated, as we had to use the UltraLibrarian to get the footprints for the components and we had to make sure that everything was connected properly.

> PCB Specs
>
> # PCB Specs
>
> TOP + BOT
>
> dim: 50.03mm x 25mm
>
> punem originea in contul placii cand facem OUTLINE-ul
> gridul initial sa fie de 2.54mm sau diviziuni cu 2 ale lui (ex. 2.54 / 2 sau / 4)
>
> initial facem outline, apoi gridul, apoi punem originile
>
> via: padshape: 0.6mm
> hole shape: 0.3mm
>
> ```
> ideal ar fi sa folosim cat mai putine via-uri
> ```

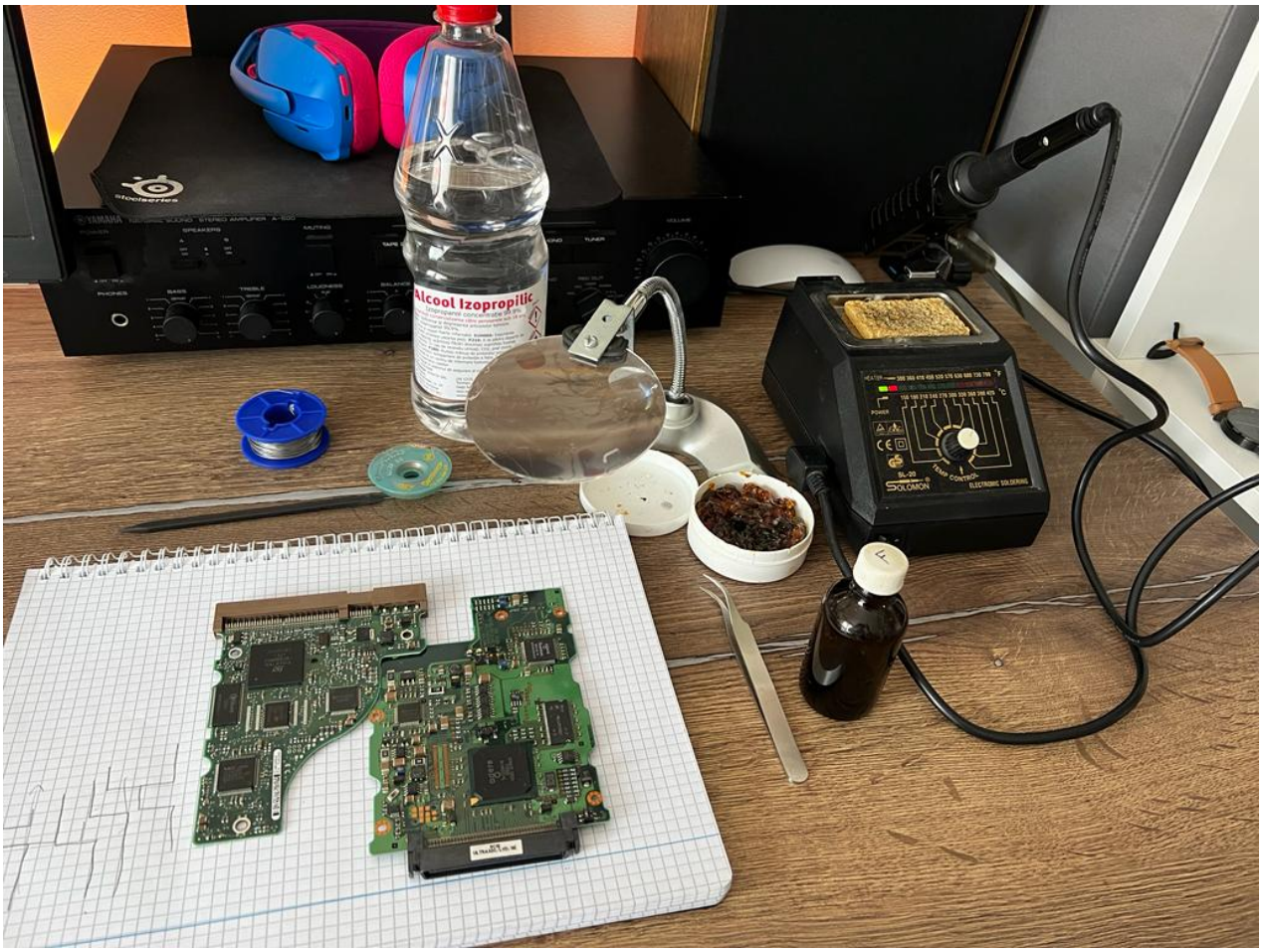
> power trace: 0.5mm
> signal trace: 0.254mm
>
> sa ne uitam la footprint-ul de la USB sa aiba liniuta care coincide cu marginea placii daca nu o avem ,desenam noi linia in footprint si o luam din datasheet

Anyhow, we managed to finish the PCB layout and we sent it to the mentors for verification. It was okay! Here is a picture from the process:



Day 7 - Soldering

Everything is ready! Let's get soldering everything on the PCB! Having used small components, this was a bit more complicated, but we managed to do it!



07-28

HDL - Day 1

Project structure

- **system_constr.xdc**: in constraints, we tell the FPGA pins and what will be connected there. We also put the delay/timing between clock and data, for example
- **system_top.v** *(v = verilog)*: here, we specify the direction of the signals, similar to a wrapper over everything we write. We know that this goes in and that this goes out etc. We divide this module into IPs
- **system_bd.tcl**: facem legaturi intre module. daca avem un modul care are 2 iesiri si unul care are 2 intrari, facem legaturile intre ele. aici facem si legaturile cu pinii de pe FPGA. asta e un script pe care il folosim in loc de GUI-ul din Vivado
- **system_bd.tcl**: we make the connections between modules. Should we have a module with 2 outputs and one with 2 inputs, then here we make the connections between these two. Here we also make the connections with the FPGA pins. This is a script that we use instead of the GUI from Vivado, which makes it a lot easier to setup the project

Tasks

Read the following:

1. HDL Coding Guideline
2. Verilog tutorial
3. Tcl tutorial
4. Makefile tutorial
5. SPI Protocol

... and install TerosHDL

07-31

HDL - Day 2

Tasks

1. Make the IP
2. Generate PWM that is proportional to the ADC

Some notes

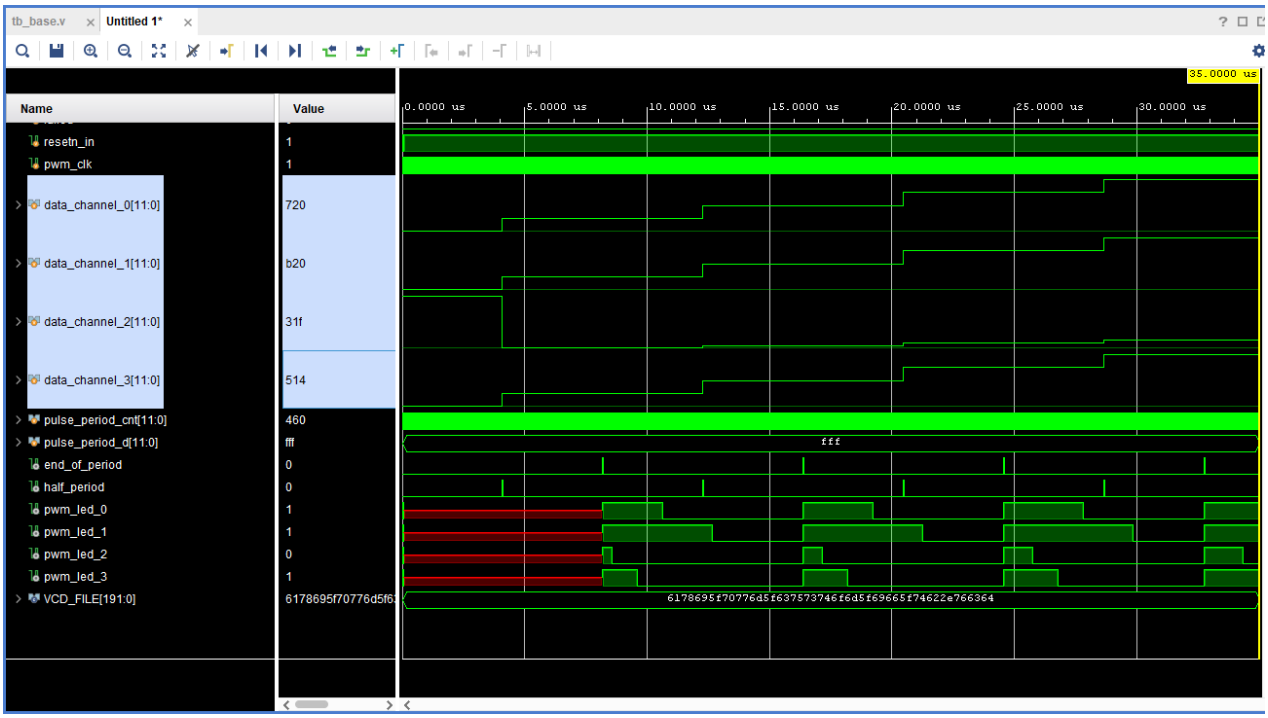
IP

To make the IP, it was not that difficult, because the mentors helped us a lot. Theoretically, we should have made a new project, add the IP, configure it, generate it, and then add it to our project. We also should have made a new block design, add the IP in there as well, and connect it to the Zynq processor. We also should have added the AXI GPIO IP, to be able to read the values from the ADC.

Thankfully, the project was already set up, the script were... scripting their way through and everything was going smoothly. We just had to make some modifications and we were done.

PWM

Here, things got a little more difficult, in the sense that we had to make a PWM generator and make a testbench for it, then analyze the results. We had to make a PWM generator that would take as input a number between 0 and 4059 and output a PWM signal with a duty cycle proportional to the input. Here's a screenshot of the result:



Pretty cool, right? This ended up as way easier than expected.

08-01

HDL - Day 3

We've been given a template to make it easier for us to make the IP. We made the necessary changes and we ran make 100 times. At some point, it worked 🙄. Therefore, we also touched a bit on Makefiles.

Tasks

1. Complete the given template
2. Build the project
3. Generate BOOT.BIN

Tip of the day

Don't hit `make` unless you checked your code. It's not fun to make a project 100 times and have it fail every time because of a typo!

08-02

HDL - Day 4

We learned that the majority of the commits are something like:

- "asd"
- "merge"
- "mergeeee"
- "final"
- "alt final"

... and stuff like that, so we needed to redo all the commits.

In the end, all of us had 4 commits. The mentors taught us how to write the "perfect" commit message. We got hands-on experience with git rebase and git reset.

08-03

HDL - Day 5

Today, we did a recap of the previous days. We touched a bit on the GUI of Vivado and we've seen all the connections between the modules. Nothing much today, it was more of an outro to the HDL part of the project.

Oh, yeah, the project

I almost forgot! The mentors also made a cool presentation of the project and we've seen how everything works together. It was pretty cool, I must say. I'm really excited to see the final product! It gives a glance on how production works vs how we do things, which is important for us, as students.

08-04

Linux Drivers - Day 1

Today we learned how an embedded system boots, from the ZSBL to the Linux kernel. We also learned about the device tree and some basic concepts regarding this topic.

Questions

1. How does the board boot?
2. What are the boot files?
3. What are the boot stages?
4. Kernel role?
5. What is ROOTFS?
6. What does the BOOT.BIT contain and WHY?

Answers

1. Mentor said: *"Tricky stuff"*
2. BOOT.BIN, image.ub, devicetree.dtb
3. Bootloaders:
   ZSBL (zero stage boot loader): loads BOOT.BIN (FSBL.elf, uBoot.elf, optional *.bit)
   FSBL (first stage boot loader): loads the PS (ii configureaza clock-urile si perifericele)
   SSBL (second stage boot loader): loads kernel, devicetree.dtb, fs

de citit despre: sys boot dev proc

08-06

Linux Drivers - Day 2

- Recap of day 1 - this took a lot! (but it was worth it)
- Environment setup
  - install VirtualBox
  - get Debian image
  - do some setup
  - test some sample code and see if eveything works

08-07

Linux Drivers - Day 3

Environment setup

Download

- gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz
- iio_rootfs.zip

Set cross compiler and architecture

```
analog@debian:~/Documents/linux_emu$ export
CROSS_COMPILE=/home/analog/Downloads/gcc-arm-10.3-2021.07-x86_64-arm-none-
linux-gnueabihf/bin/arm-none-linux-gnueabihf-

analog@debian:~/Documents/linux_emu$ export ARCH=arm
```

Generate config

```
analog@debian:~/Documents/linux_emu$ make versatile_adi_defconfig
```

Generate device tree

```
analog@debian:~/Documents/linux_emu$ make versatile-pb-adi-emu.dtb
```

Build

The virtual machine has 4 cores. I used `-j4`, but it broke down, so I reduced it to `-j3`. All good now!

```
analog@debian:~/Documents/linux_emu$ make -j3
```

Run

```
analog@debian:~/Documents/linux_emu$ qemu-system-arm \
-machine versatilepb \
-kernel arch/arm/boot/zImage \
-dtb arch/arm/boot/dts/versatile-pb-adi-emu.dtb \
-drive file=/home/kali/Downloads/iio_rootfs/rootfs.ext2,if=scsi,format=raw \
-append "root=/dev/sda console=ttyAMA0,115200" \
-serial stdio -net nic,model=rtl8139 -net user
```

Login with `root` and `analog`. Done! We are in the emulation environment.

Starting to write the driver

- open Visual Studio Code in the linux_emu folder
- install the C/C++ extension pack
- configure the editor with settings.json from mentor (very important)
- create a new file named `iio_adi_emu.c`
- start writing!

Kernel coding style

- the kernel wants to have maximum 80 columns
- the kernel wants to have tab size 8, no spaces
- the upstream wants a blank line between certain includes

To make things easier, the mentor got us this nice settings.json:

```
{
    "[c]": {
        "editor.detectIndentation": false,
```

```
        "editor.tabSize": 8,
        "editor.insertSpaces": false,
        "editor.rulers": [80]
    },
    "C_Cpp.default.includePath": [
        "${workspaceFolder}/**",
        "${workspaceFolder}/include",
        "${workspaceFolder}/arch/arm/include",
        "${workspaceFolder}/arch/arm/include/generated"
    ],
    "C_Cpp.default.defines": [
        "__KERNEL__"
    ],
    "C_Cpp.default.cStandard": "c11",
    "C_Cpp.default.cppStandard": "c++17",
    "C_Cpp.default.intelliSenseMode": "linux-gcc-arm",
    "C_Cpp.default.configurationProvider": "ms-vscode.makefile-tools",
    "files.associations": {
        "random": "c"
    }
}
```

08-08

Linux Drivers - Day 4

Let's test the Zybo board

- change root's pwd: `parolanoua`
- change the board's mac addr: add `ethaddr=<MAC-ul nou>` in `/boot/uEnv.txt`
- get its ip: `ip a`
- connect to it: `ssh root@<ip>`

Done!

Should one not see the device in `/sys/bus/iio/devices`

Possible reasons:

- the code is not okay
- the device tree is not okay
- it's not selected in menuconfig

Let's make a recap

Kernel stuff

- downloaded the kernel
- made `driver.c` (aka ad5592r_s.c)
- modified the `Kconfig` (which makes the entry in menuconfig)
- modified the `Makefile`
- created the `uImage` (which is, in fact, the kernel itself)

Device tree stuff

- created `file.dts` (aka zynq-zybo-z7-ad5592r.dts)
- used a Makefile to generate the blob `file.dtb` (aka zynq-zybo-z7-ad5592r.dtb)
- renamed `zynq-zybo-z7-ad5592r.dtb` to `devicetree.dtb`

Zybo-related stuff

- made the SD Card using Rufus
  - it has 3 partitions:
    - **rootfs** (not visible from Windows)
    - **BOOT** (here we have BOOT.BIN, uImage and devicetree.dtb)
    - an empty partition
- copied the uImage and devicetree.dtb to the BOOT partition
- rebooted the board

linux-last

Linux Drivers - Days 5, 6 and 7

Day 5

Today, we implemented functions which would read and write to and from the device. These were the most important functions, as they are the ones which will be used the most. These functions will pe used by the user to read and write to the device. Here they are:

```
static int ad5592r_s_read_raw(struct iio_dev *indio_dev,
                              struct iio_chan_spec const *chan,
                              int *val,
                              int *val2,
                              long mask)
{
        struct ad5592r_s_state *st = iio_priv(indio_dev);
        int ret;
```

```
        switch (mask)
        {
        case IIO_CHAN_INFO_RAW:
                ret = ad5592r_s_spi_read_adc(st, chan->channel, val);
                if (ret) {
                        dev_err(&st->spi->dev, "Failed reading from chan:
%d\n", chan->channel);
                        return ret;
                }
                return IIO_VAL_INT;

        case IIO_CHAN_INFO_ENABLE:
                *val = st->en;
                return IIO_VAL_INT;

        default:
                return -EINVAL;
        }
}

static int ad5592r_s_write_raw(struct iio_dev *indio_dev,
                               struct iio_chan_spec const *chan,
                               int val,
                               int val2,
                               long mask)
{
        struct ad5592r_s_state *st = iio_priv(indio_dev);
        switch (mask)
        {
        case IIO_CHAN_INFO_ENABLE:
                st->en = val;
                return 0;

        default:
                return -EINVAL;
        }
}
```

Day 6

Now, we added debugfs support. This is a very useful feature, as it allows us to see the values of the registers and the values of the channels. Moreover, we can also change the values of the registers and the channels.

We also did some testing and debugging, and we found out some bugs that we're gonna fix tomorrow.

Day 7

As previously mentioned, the day started with fixing some issues. Then, we started reading and understanding the datasheet of the convertor better, so that we can implement the rest of the functions. We also started implementing the functions for the rest of the channels + testing and debugging.

08-10

Linux Drivers - Day 8

For the last part of the Linux drivers module, we implemented a buffer, so that we can read with a greater precision and at a constant rate (almost "guaranteed" thing). This way, we minimise the user space timing errors that can occur because of the scheduler and different priorities.

To do this, we modified some parts of the code and, in essence, we just optimised the driver and made the last step there was to make (at least for what the practice program was concerned). We hand-tested the buffer and made a really cool recap of everything, than we discussed with the mentors and everything was great.

08-21

Day 1 - Apps

To get a kickstart on the apps module, we started by getting a glimpse of **bash scripts**. I have writted a cool script which would write values from 1 to n to the AD5592R channels 0 to 5. Here it is:

```
#!/bin/bash

# Help function
Help()
{
    echo "Usage: $0 [-i <ip>] [-n <n>]"
    echo "Example: $0 -i 10.76.84.153 -n 10"
}

# About function
About()
{
    echo "This script writes values '1 to n' to AD5592R channels '0 to 5'."
    echo "  Targer IP: $1"
    echo "  Number of values to write: $2"
    echo
}
```

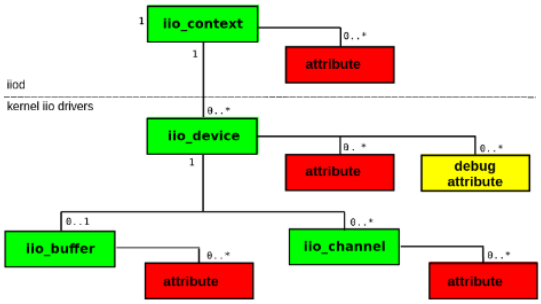
```
# Check if there are 4 arguments
if [ $# -ne 4 ]; then
    Help
    exit 1
fi

# Check if the user needs help
if [ $1 == "-h" ] || [ $1 == "--help" ]; then
    Help
    exit 0
fi

# Parse command line arguments
while getopts ":i:n:" option; do
    case $option in
        i) # IP address
            ip=$OPTARG
            # echo "IP: $OPTARG"
            ;;
        n) # number of values to write
            n=$OPTARG
            # echo "n: $OPTARG"
            ;;
        ?) # unknown option
            echo "Error: Unknown option: -$OPTARG"
            Help
            exit 1
            ;;
    esac
done

About $ip $n

for i in $(seq 1 $n)
do
    # Write to channels 0 to 5
    for chan in $(seq 0 5)
    do
        OUTPUT=$(iio_attr -u ip:${ip} -c ad5592r_s voltage${chan} raw ${i})
        echo "Writing to channel ${chan} value ${OUTPUT}"
    done

    echo

    # Delay of 1 second
    sleep 1
done
```

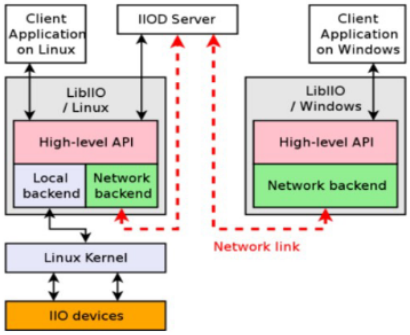
Then, we got to do some simple **C** apps. Hello world, then an IIO Context Hello World and then some more complicated stuff. The next day should be more promising!

08-22

Day 2 - Apps

Today, it was all about understanding how everything works together and communicates. We took a look at the following diagrams:





After that, we started to learn about CMake and got to generate some neat makefiles. After that, we did some testing with `iio_<app>` command line interface apps, to better understand IIO hierarchy. This will help us in the following day, when we will start to write our own apps.

08-23

Day 3 - Apps

The challenges of the day were:

```
1. Scrieti programul C "read_adc" care citeste de la ADC valorile de raw ale
axelor xpoz,xneg si le afiseaza in urmatorul format:

xpoz <raw> <volts> <acceleration>
xneg <raw> <volts> <acceleration>
ypoz <raw> <volts> <acceleration>
yneg <raw> <volts> <acceleration>
zpoz <raw> <volts> <acceleration>
zneg <raw> <volts> <acceleration>

2. Dupa ce avem programul care ne afiseaza xpoz,xneg, etc .. in raw,volti, si
g (acceleratie gravitationala), folositi M2k + Scopy pentru a masura iesirea
de la opamp. Validati ca valoarea de volti obtinuta in programul C "" este
egala cu iesirea de la opamp.
```

```
3. Scrieti un program "read_accel", care afiseaza X,Y,Z in bucla
X: <acceleration> Y:<acceleration> Z:<acceleration>

4. Scrieti un program "calib_accel" care ne ofera instructiuni pentru
calibrarea placii:"Place the module on a plane surface"

<read user input>
"Adjust the potentiometers according to instructions"
"POT1: clockwise POT2: counter-clockwise POT3: center" (example)
"POT1: clockwise POT2: counter-clockwise POT3: center"
"POT1: clockwise POT2: center POT3: center"
"POT1: center POT2: center POT3: center"
"Calibrated !"

5. Scrieti un program "shock_detect" care asteapta producerea unui soc. In
urma socului programul va afisa "<timestamp> - shock detected"
```

Look at this output!

idx	addr	x+ ch1	x- ch2	y+ ch3	y- ch4	z+ ch5	z- ch6
0	56321a1f4830	250	9	221	14	2266	1
1	56321a1f483c	249	9	190	14	2262	1
2	56321a1f4848	248	9	226	13	2253	1
3	56321a1f4854	253	9	194	14	2248	1
4	56321a1f4860	250	9	195	14	2228	1
5	56321a1f486c	253	9	197	13	2216	1
6	56321a1f4878	256	9	204	14	2207	1
7	56321a1f4884	253	9	199	13	2193	0
8	56321a1f4890	252	9	199	13	2178	1
9	56321a1f489c	248	9	199	14	2172	1
10	56321a1f48a8	248	9	197	14	2162	1
11	56321a1f48b4	248	9	200	14	2162	1
12	56321a1f48c0	245	10	192	14	2166	1
13	56321a1f48cc	250	9	191	14	2171	1
14	56321a1f48d8	253	9	191	13	2186	1
15	56321a1f48e4	254	9	186	14	2194	1
16	56321a1f48f0	255	9	197	14	2206	0
17	56321a1f48fc	252	9	198	14	2219	1
18	56321a1f4908	250	9	196	14	2225	1
19	56321a1f4914	251	9	198	14	2229	1
20	56321a1f4920	250	9	197	14	2226	1
21	56321a1f492c	251	9	201	14	2231	1
22	56321a1f4938	251	9	197	14	2228	1
23	56321a1f4944	252	8	193	14	2214	0
24	56321a1f4950	251	8	200	13	2226	1
25	56321a1f495c	254	8	200	13	2212	0
26	56321a1f4968	261	9	202	14	2209	1
27	56321a1f4974	259	9	201	14	2208	1
28	56321a1f4980	257	9	203	13	2208	0
29	56321a1f498c	258	8	198	13	2218	1
30	56321a1f4998	252	8	203	13	2225	0
31	56321a1f49a4	256	9	202	13	2237	1

08-24

Day 4 - Apps

We continued the challenges of the previous day and then, after all of them were completed, we started to work on the final app, which was a Python application with graphical interface.

This app used `matlibplot` to draw a level indicator, which was updated in real time, based on the accelerometer data. Here is the code and the result:

```
import numpy as np
import iio
import matplotlib.pyplot as plt

ctxname = "ip:10.76.84.240"
channel_prefix = "voltage"
nb_channels = 6

def getRaw(channel):
    return int(channel.attrs["raw"].value)

# Get context and device
ctx = iio.Context(ctxname)
dev = ctx.find_device("ad5592r_s")

channels = []
for i in range(6):
# Get all channels
    channels.append(dev.find_channel("{}{}".format(channel_prefix, i)))

while True:
    x = getRaw(channels[0]) - getRaw(channels[1])
    y = getRaw(channels[2]) - getRaw(channels[3])
    z = getRaw(channels[4]) - getRaw(channels[5])

    if z < 0:
        z = np.array([z, 0])
    else:
        z = np.array([0, z])

    plt.bar(2000, z, width = 500, color = "mediumaquamarine")
    plt.plot(x, y, 'o', color = "mediumpurple")
    plt.axis([-4096, 4096, -4096, 4096])
    plt.pause(0.05)
    plt.clf()
```


Creator
Checking...
Figure 1
File Actions Edit View Help
DONT UPLOAD
TO
VIRUS TOTAL
PLEASE DON'T UPLOAD BACKDOOR TO WWW.VIRUSTOTAL.COM
YOU CAN UPLOAD OUTPUT/BACKDOOR FILE TO WWW.NODISTRIBUTE.COM
RECON
LOOT
PAYLOAD

Thank you ADI

🌟 A Journey Beyond Just Electronics

Throughout the six-week program with Analog Devices, I embarked on a profound journey that transcended boundaries, challenging me to delve into various technical domains.

✨ From Breadboards to Bits & Bytes

From revisiting fundamental electronics to understanding the intricacies of advanced software development, this program provided a holistic learning experience.

💡 Mentors: The Guiding Lighthouses

A heartfelt thank you to the mentors at Analog Devices. Your expertise, patience, and passion illuminated my path, ensuring I never felt lost, even in the most complex phases of the project.

🔗 Bridging Domains: An Interdisciplinary Wonderland

This experience wasn't just about individual components or isolated software modules. It was about understanding how these diverse domains intricately weave together to create coherent systems. This realization will undoubtedly shape my approach to future projects.

🙏 In Closing...

To the entire team at Analog Devices and all my peers, thank you for making this journey enlightening, challenging, and immensely rewarding. The skills and perspectives I've gained are invaluable, and I'm excited to apply them in future endeavors.


Champion
CERTIFICATE OF COMPLETION