CSE 121: Lab 2 Report

By Daniel Beltran

10/20/2023

Lab 2: Part 1

The first portion of the lab proved to be quite challenging as utilizing GDB without source code was new for myself. The first instance was creating a project directory and downloading the .elf file, provided by the staff. From the lab2_debug.elf file, I was able upload the image and grab the .bin file generated by the following command:

Bin Source:

https://docs.espressif.com/projects/esptool/en/latest/esp32c3/esptool/basic-commands.html

From this, the next step was to flash the files and run GDB. Unfortunately, it wasn't that simple as using the `idf.py flash` will effectively overwrite all files within the build directory. What does this mean and how does the setup work? Essentially, I would have to find another way to flash my files and along with making sure the initial setup (e.g. when using ./install.sh esp32c3, . export.sh, idf.py set-target esp32c3) as it tended to also overwrite the build directory. Each initial setup, I would have to remove the binary and elf files from the build directory to prevent any deletion. In order bypass this flashing overwritting, I used this command:

```
openocd -f board/esp32c3-builtin.cfg -c "program_esp filename.bin 0x10000 verify exit"
```

Flash Source:

https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/api-guides/jtag-debugging/index.html

However, before we can even flash there must be a .gdbinit file in place so that we can connect to port:3333 when calling on idf.py openocd. Straight from the lab documentation, these following lines of code will be necessary initiaing a client/server connection (e.g. board to gdb). Below will show the lines added onto the .gdbinit file:

```
target remote :3333
set remote hardware-watchpoint-limit 2
mon reset halt
flushregs
b app_main
c
```

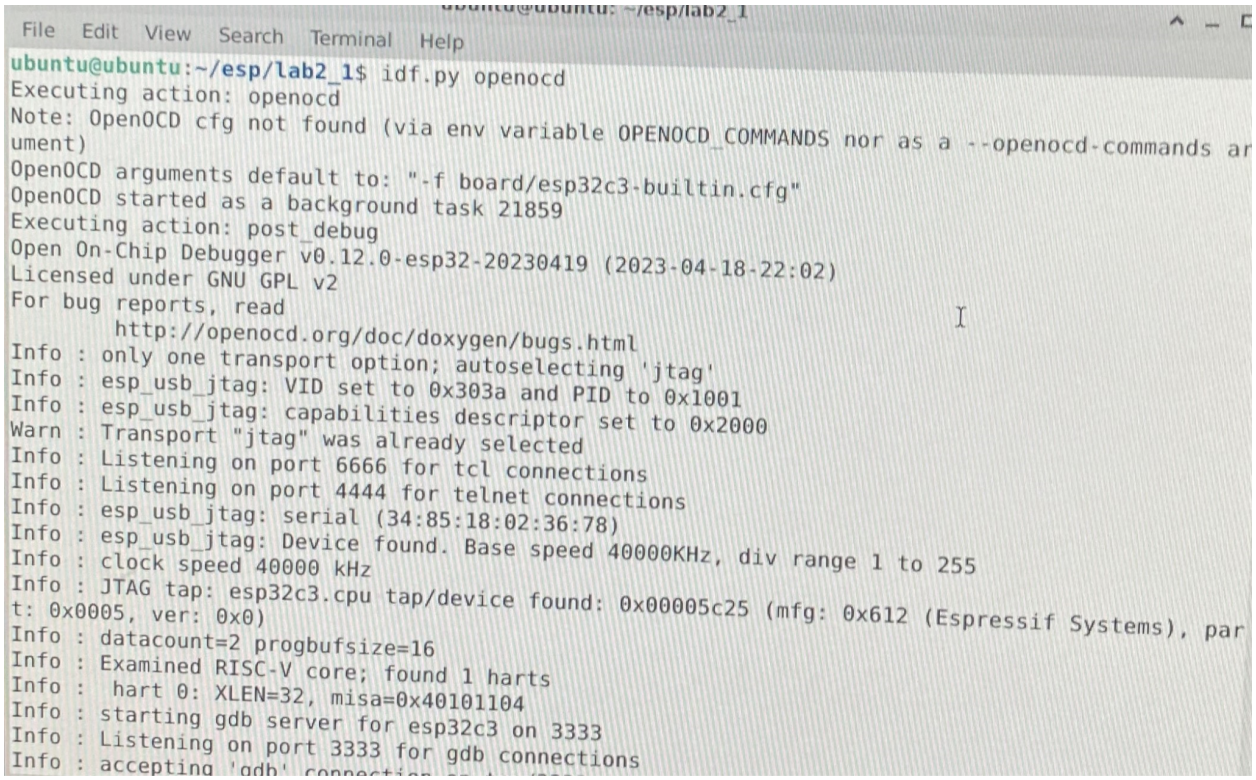
Another step that must be taken before getting into GDB is setting up the udev rules, pointed out by the staff. Essentially, I had to add these set of rules to properly setup my JTAG. The following directory is where the udev rules were added onto:

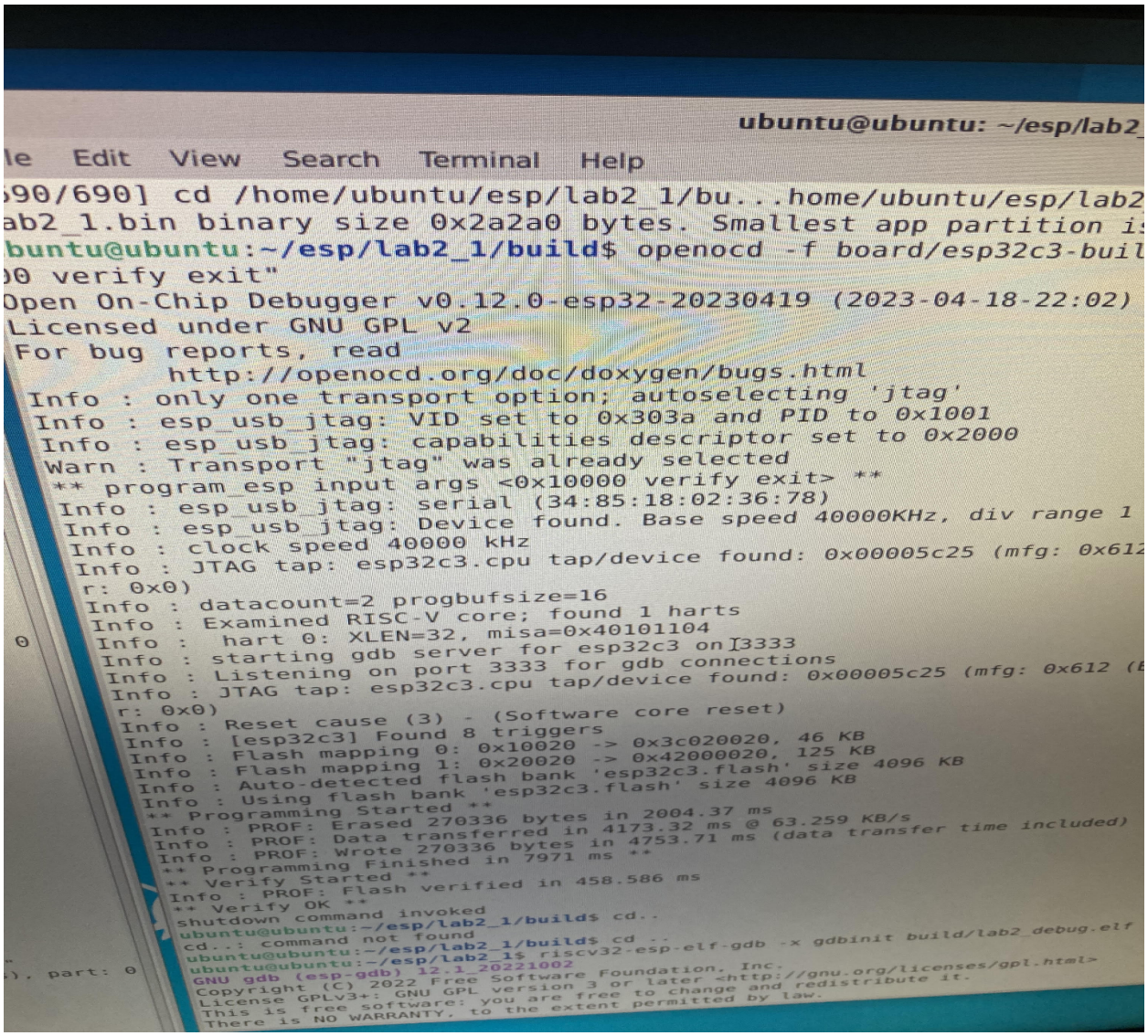
```
/etc/udev/rules.d
```

Udev Source:

https://docs.espressif.com/projects/esp-idf/en/latest/esp32c3/api-guides/jtag-debugging/configure-builtin-jtag.html

After adding these set of rules it was time to start the debugging process. One thing that held me back many times was the fact that two terminals were needed to: flash the bin file, establish the client/server connection, and run the RISCV command (riscv32-esp-elf-gdb -x gdbinit build/lab2_debug.elf, given by the professor).



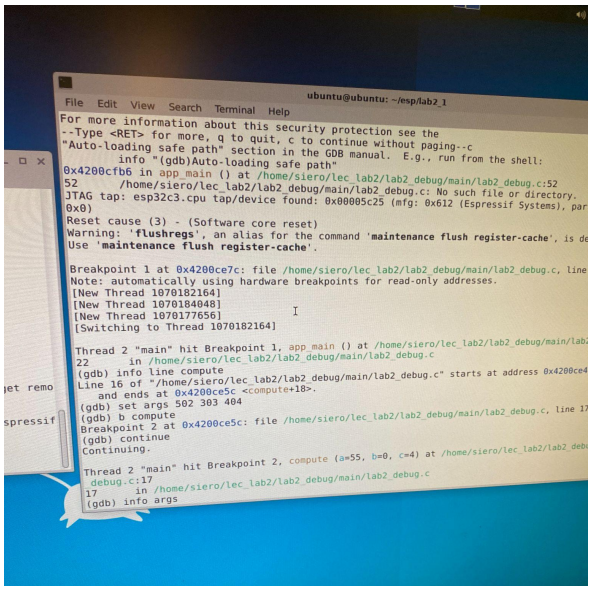


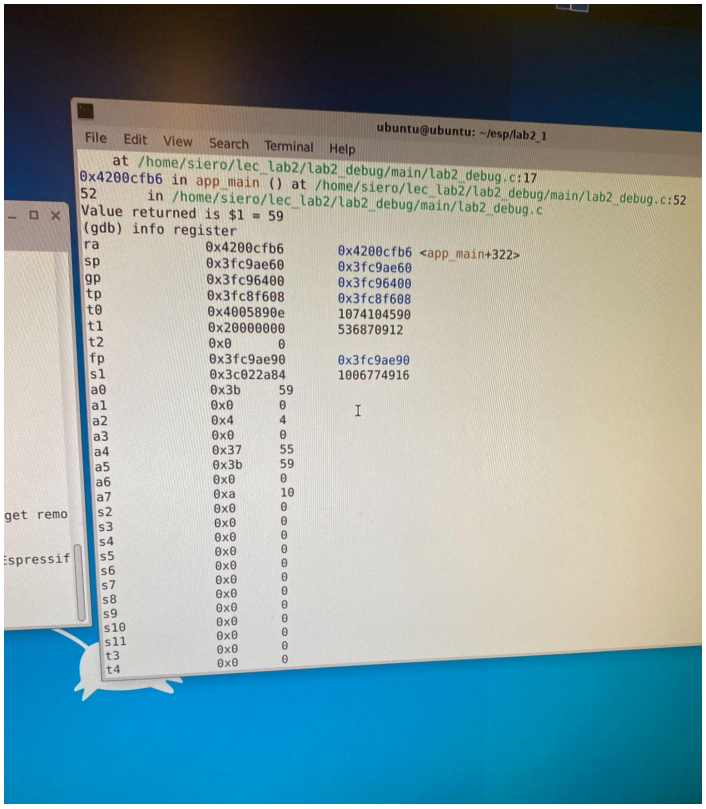
The pictures above show two different terminals that provide us with the client/server connection and running of GDB. Once into GDB it was quite confused until I had read on the different commands that can be used. The command “info registers” or “i r” showed a detailed description of all the registers within and what caught my attention was the compute function. However, that was merely to assist in displaying the registers and printing out their addresses, but how would we return inputs from a function? The use of “set args [arguements]” assisted in providing an

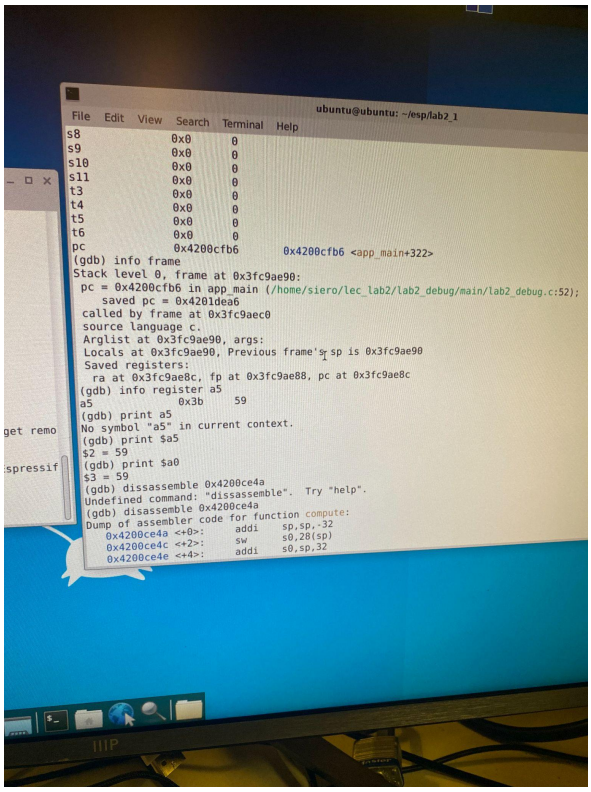
input to the compute function. Taking in the values, given in the lab documentation (502, 303, 404), I was able to successfully input the following commands:

```
set args 502 303 404

b compute

Continue

info args

finish
```

These set of commands gives us the outputs of a = 55(0x37), b = 0(0x0), and c = 4(0x4). These values correspond to the integer inputs of the function compute. The list of commands are done this way since we are only provided an .elf file. The return address is 0x4200ce72 of the return value 59. The entry of the computer function is 00x4200ce5c. Below will be a list of photos detailing what is mentioned, previously.




ubuntu@ubuntu: ~/esp/lab2_1
File Edit View Search Terminal Help
at /home/siero/lec_lab2/lab2_debug/main/lab2_debug.c:17
0x4200cfb6 in app_main () at /home/siero/lec_lab2/lab2_debug/main/lab2_debug.c:52
52 in /home/siero/lec_lab2/lab2_debug/main/lab2_debug.c
Value returned is $1 = 59
(gdb) info register
ra 0x4200cfb6 0x4200cfb6 <app_main+322>
sp 0x3fc9ae60 0x3fc9ae60
gp 0x3fc96400 0x3fc96400
tp 0x3fc8f608 0x3fc8f608
t0 0x4005890e 1074104590
t1 0x20000000 536870912
t2 0x0 0
fp 0x3fc9ae90 0x3fc9ae90
s1 0x3c022a84 1006774916
a0 0x3b 59
a1 0x0 0
a2 0x4 4
a3 0x0 0
a4 0x37 55
a5 0x3b 59
a6 0x0 0
a7 0xa 10
s2 0x0 0
s3 0x0 0
s4 0x0 0
s5 0x0 0
s6 0x0 0
s7 0x0 0
s8 0x0 0
s9 0x0 0
s10 0x0 0
s11 0x0 0
t3 0x0 0
t4 0x0 0


ubuntu@ubuntu: ~/esp/lab2_1
File Edit View Search Terminal Help
s8 0x0 0
s9 0x0 0
s10 0x0 0
s11 0x0 0
t3 0x0 0
t4 0x0 0
t5 0x0 0
t6 0x0 0
pc 0x4200cfb6 0x4200cfb6 <app_main+322>
(gdb) info frame
Stack level 0, frame at 0x3fc9ae90:
pc = 0x4200cfb6 in app_main (/home/siero/lec_lab2/lab2_debug/main/lab2_debug.c:52);
saved pc = 0x4201dea6
called by frame at 0x3fc9aec0
source language c.
Arglist at 0x3fc9ae90, args:
Locals at 0x3fc9ae90, Previous frame's sp is 0x3fc9ae90
Saved registers:
ra at 0x3fc9ae8c, fp at 0x3fc9ae88, pc at 0x3fc9ae8c
(gdb) info register a5
a5 0x3b 59
(gdb) print a5
No symbol "a5" in current context.
(gdb) print $a5
$2 = 59
(gdb) print $a0
$3 = 59
(gdb) dissassemble 0x4200ce4a
Undefined command: "dissassemble". Try "help".
(gdb) disassemble 0x4200ce4a
Dump of assembler code for function compute:
0x4200ce4a <+0>: addi sp,sp,-32
0x4200ce4c <+2>: sw s0,28(sp)
0x4200ce4e <+4>: addi s0,sp,32

GDB Commands: https://web.cecs.pdx.edu/~apt/cs491/gdb.pdf

Set Args: https://visualgdb.com/gdbreference/commands/set_args

Lab 2: Part 2

After a long struggle with the first portion of the lab, first started researching what parts of the ESP32C3 board we'd be using and noticed that the SHTC3 sensor is built in within. Essentially, it gave me an idea of what to look for and used ChatGPT to start off the code. Essentially, I had given it the information of what type of board we'd be using, the protocol of I2C, and the functionality that is looked for, temperature and humidity. Below will show a screenshot of the code snippet given by ChatGPT, which severed as a great starting point that needed alterations for it to work properly.

c Copy code

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/i2c.h"

#define I2C_MASTER_SCL_IO 22        // GPIO for the I2C clock line
#define I2C_MASTER_SDA_IO 21        // GPIO for the I2C data line
#define I2C_MASTER_NUM I2C_NUM_0    // I2C port number
#define I2C_MASTER_FREQ_HZ 100000   // I2C clock frequency

#define SHTC3_ADDR 0x70            // I2C address of the SHTC3 sensor

void i2c_master_init() {
    i2c_config_t conf;
    conf.mode = I2C_MODE_MASTER;
    conf.sda_io_num = I2C_MASTER_SDA_IO;
    conf.sda_pullup_en = GPIO_PULLUP_ENABLE;
    conf.scl_io_num = I2C_MASTER_SCL_IO;
    conf.scl_pullup_en = GPIO_PULLUP_ENABLE;
    conf.master.clk_speed = I2C_MASTER_FREQ_HZ;
    i2c_param_config(I2C_MASTER_NUM, &conf);
    i2c_driver_install(I2C_MASTER_NUM, conf.mode, 0, 0, 0);
}
```

```
esp_err_t read_temperature_and_humidity(float *temperature_C, float *humidit
    uint8_t data[6];
    i2c_cmd_handle_t cmd = i2c_cmd_link_create();

    i2c_master_start(cmd);
    i2c_master_write_byte(cmd, (SHTC3_ADDR << 1) | I2C_MASTER_READ, 1);
    i2c_master_read(cmd, data, 6, I2C_MASTER_LAST_NACK);
    i2c_master_stop(cmd);

    esp_err_t ret = i2c_master_cmd_begin(I2C_MASTER_NUM, cmd, 1000 / portTIC
    i2c_cmd_link_delete(cmd);

    if (ret != ESP_OK) {
        return ret;
    }

    // Replace the following with the correct data byte indices and conversi
    uint16_t raw_temp = (data[0] << 8) | data[1];
    uint16_t raw_humidity = (data[3] << 8) | data[4];

    *temperature_C = -45.0 + 175.0 * (float)raw_temp / 65535.0;
    *humidity = 100.0 * (float)raw_humidity / 65535.0;

    return ESP_OK;
}
```

```
void app_main() {
    float temperature_C;
    float humidity;

    i2c_master_init();

    while (1) {
        if (read_temperature_and_humidity(&temperature_C, &humidity) == ESP_O
            // Process and print the temperature and humidity data
        } else {
            printf("Failed to read data from the sensor\n");
        }

        vTaskDelay(2000 / portTICK_RATE_MS);
    }
}
```

The main source of difficulty was figuring out some issues with the code given by generative AI.

One thing that was causing the program to exit into the else statement was the fact that the AI only provided me with writing into the file and not reading. Through much of the trial and error, I had to input the address values of 0x7CA2, a 16-bit value representing the reading of temperature with the sensor SHTC3. We pass this in the second parameter of `i2c_master_write` for the sole purpose of ensuring that the temperature is being read and written out on the terminal. Though, provided with starting code by ChatGPT assisted in the learning process of this part of the lab. Looking over Github repositories of the I2C protocol, it showed me how to utilize different functions along with the proper syntax from using the "driver/i2c.h" header file. Below will show the sources of what was stated.

SHTC3:https://sensirion.com/media/documents/643F9C8E/63A5A436/Datasheet_SHTC3.pdf

I2C Protocol Repo:

https://github.com/espressif/esp-idf/blob/master/examples/peripherals/i2c/i2c_simple/main/i2c_simple_main.c