Lab 2 Guide

lab2.md
File

This is the file that describes a lab I need to do. Please go through the lab document, determine the goals and requirements to complete the lab, and let me know how I should go about completing the lab. If you have any questions about hardware, software, or any other details that should be clarified for you to give me a thorough, complete answer, please ask.

Based on the lab document, here are the goals and requirements for Lab 2 and a suggested approach to complete it.

Lab Goals and Requirements:

Lab2.1: Debugging with GDB (10 points)

1. **Objective**: Use GDB to debug a RISC-V executable (`lab2_debug.elf`), specifically to:
    - Upload the binary to an ESP32 board.
    - Use GDB to debug the binary and figure out the values passed to a function called `compute` .
    - Identify the entry point address of the `compute` function.
    - Identify the address of the return instruction within the `compute` function.
2. **Requirements**:
    - Convert `lab2_debug.elf` to `.bin` and upload it to the ESP32.
    - Run GDB using `lab2_debug.elf` .
    - Obtain the arguments and return value for the `compute` function.
    - Find the entry point and return address of the `compute` function.
    - Demonstrate this during the check-off time using GDB.
3. **Steps for Completion**:
    - Use `idf.py openocd` to enable communication with the ESP32 board.
    - Use GDB (`riscv-esp32-elf-gdb`) along with a `gdbinit` file to configure debugging settings.

- Flash the binary using OpenOCD and set breakpoints appropriately.
- Analyze the assembly code to identify where values are being passed and how the return value is calculated.
- Learn basic GDB commands (e.g., `b`, `info registers`, `x/i`, `stepi`) to step through the program and inspect register values.
- Document the process clearly in `report.pdf`.

Lab2.2: Humidity and Temperature Sensor (10 points)

1. **Objective**: Interface with the SHTC3 temperature and humidity sensor on the ESP32 board to:
   - Write a program that reads the temperature and humidity every 2 seconds.
   - Print temperature in both °C and °F, and humidity as a percentage.
2. **Requirements**:
   - Write separate functions for reading temperature and humidity, each reading at most 3 bytes.
   - Use power-up and power-down sequences between reads.
   - Use the checksum to verify data.
   - The program should print readings in a specific format every 2 seconds.
3. **Steps for Completion**:
   - Implement the I2C communication to interact with the SHTC3 sensor.
   - Create functions to:
     - Power up the sensor.
     - Read temperature and humidity values.
     - Power down the sensor.
   - Convert temperature values to both °C and °F.
   - Format and print the output every 2 seconds.
   - Use a checksum to verify data validity after each read.
   - Test your implementation on the ESP32 to ensure readings are correct.

Suggested Approach to Complete the Lab:

1. **Set Up Environment**:

- Ensure the ESP32 toolchain is installed correctly, including OpenOCD and GDB.
- Install necessary software dependencies (e.g., `idf.py`).
- Review the RISC-V function calling convention for better understanding of argument passing.

2. **Lab2.1 Steps**:
   - Use `riscv64-unknown-elf-objdump` to disassemble `lab2_debug.elf` into assembly code (`lab2_image.S`).
   - Flash the `lab2_debug.bin` to the ESP32 board.
   - Start OpenOCD (`idf.py openocd`) to enable communication.
   - Use `riscv-esp32-elf-gdb` to connect to the board and debug using GDB.
   - Set breakpoints, step through the code, and examine registers to determine:
     - The arguments and return value of the `compute` function.
     - The entry point and return address of the `compute` function.
   - Document the process and results in the required `report.pdf`.
3. **Lab2.2 Steps**:
   - Write a new program that initializes I2C communication with the SHTC3 sensor.
   - Create separate functions for reading temperature and humidity, including power-up/power-down logic.
   - Use a timer or delay function to ensure the readings are spaced 2 seconds apart.
   - Implement data verification using the checksum.
   - Print the temperature in both °C and °F, and humidity as a percentage.
   - Test the program on the ESP32 to verify that the readings are correct and formatted as expected.
4. **Testing and Debugging**:
   - Test GDB functionality with a simple "Hello World" program before attempting `lab2_debug.elf`.
   - Verify I2C communication with the SHTC3 sensor by manually checking temperature/humidity readings for correctness.
   - Use GDB to debug both parts of the lab to identify any runtime issues.
5. **Submission**:

- Prepare the necessary files for submission, excluding the `build` directory.
- Ensure `report.pdf` is complete and clearly documents your approach, results, and justifications for Lab2.1.

Questions for Clarification:

1. Do you have the necessary hardware available, including the ESP32 board and a way to interface with the SHTC3 sensor?
2. Are you familiar with using the ESP-IDF framework (`idf.py`) for flashing the board and setting up OpenOCD?
3. Do you need any additional support or resources for using GDB with RISC-V executables?

Let me know if you need further details or assistance on any specific part of this lab.

> Let's focus on 2.1 for now. How do I go about debugging with GDB?

To complete **Lab 2.1**, you'll need to debug the provided RISC-V executable (`lab2_debug.elf`) using GDB on your ESP32 board. Here's a step-by-step approach to guide you through the debugging process with GDB:

Step-by-Step Debugging with GDB

1. **Prepare Environment and Convert ELF to BIN**
   - Start by converting the provided ELF file to a binary file that can be flashed to the ESP32 board.
   - Use the following command to convert the ELF to BIN:

     ```bash
     riscv64-unknown-elf-objcopy -O binary lab2/lab2_debug.elf lab2_debug.bin
     ```

2. **Flash the Binary to the ESP32**
   - Before flashing, ensure OpenOCD is running to enable communication with the board. Start OpenOCD using the following command:

     ```bash
     idf.py openocd
     ```

- Now, flash the `lab2_debug.bin` file to the ESP32 using a tool like `esptool.py` or an equivalent method supported by ESP-IDF. Be cautious to use the correct command as some options like `idf.py flash` might rebuild the ELF, which could erase your current version.

3. **Launch GDB and Connect to the Board**
   - Start GDB using the provided ELF file, and pass in a `gdbinit` file for initial configuration:

     ```bash
     riscv-esp32-elf-gdb -x gdbinit lab2/lab2_debug.elf
     ```

   - The `gdbinit` file should contain startup commands to configure GDB:

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

     - `target remote :3333` establishes a connection with OpenOCD via port 3333.
     - `mon reset halt` resets and halts the board so that you can start debugging from a clean state.

4. **Set Breakpoints and Analyze the Code**
   - Set breakpoints to help identify where specific parts of the code execute. In this case, you need to determine:
     - The **arguments passed to the `compute` function**.
     - The **return address of the `compute` function**.
   - Use the following GDB commands to set breakpoints:

     ```gdb
     b compute
     ```

     or, if the symbol is not recognized, you can set a breakpoint by address:

```gdb
b *0x4200bbcc  # Replace 0x4200bbcc with the actual address from your objdump analysis.
```

5. **Disassemble the ELF File**
   - Since the ELF was compiled without debug information, you need to disassemble it to understand the code structure.
   - Use `objdump` to generate an assembly listing:

     ```bash
     riscv64-unknown-elf-objdump -D lab2/lab2_debug.elf > lab2_debug.S
     ```

   - Review the generated `.S` file to understand the function calls, entry points, and return instructions.
   - Look for the `compute` function to find its entry address and determine which registers are being used to pass arguments.
6. **Analyze Function Arguments and Return Values**
   - In the RISC-V calling convention:
     - Function arguments are typically passed in registers `a0`, `a1`, `a2`, etc.
     - Return values are usually stored in `a0`.
   - To examine these registers in GDB after breaking at the `compute` function, use:

     ```gdb
     info registers
     ```

     - Look specifically at registers `a0`, `a1`, `a2` to determine the function arguments.
     - Once the function is about to return, check `a0` for the return value.
7. **Determine the Entry and Return Addresses**
   - To find the entry point of the `compute` function, you can use the following command in GDB:

     ```gdb
     ```

```
disassemble compute
```

- This will display the disassembled code for the `compute` function, allowing you to note the entry address.

- To find the return address within the `compute` function, look for the `ret` instruction and note its address.

8. **Document Findings**
   - Record all the information about the function arguments, return value, entry address, and return address in the `report.pdf`.
   - Provide justifications for how you determined each value, including references to GDB commands used, register values checked, and analysis of the disassembled code.

GDB Commands Overview

Here are some key GDB commands that will be helpful during the debugging process:

- **Start GDB and connect**: `target remote :3333`
- **Set breakpoint**:
  - By function name: `b compute`
  - By address: `b *0x4200bbcc`
- **Continue execution**: `c`
- **Step through instructions**: `stepi`
- **Print register values**: `info registers`
- **Disassemble function**: `disassemble compute`
- **Reset the board**: `mon reset halt`

Tips for Stability

- The cross-compiled GDB and OpenOCD can be unstable. If you encounter errors such as "Connect failed" or other OpenOCD issues:
  - **Unplug and re-plug** the ESP32 board.
  - **Run** `mon reset halt` multiple times if needed to reset the board correctly.

- **Avoid pressing the physical reset button**—instead, use the software reset commands.
- **Restart GDB** if it becomes unresponsive.

Example Report Entry for `compute`

In your `report.pdf`, document findings like:

```vbnet
Lab2.1
    Compute 1st argument is 502 and is passed via register a0
    Compute 2nd argument is 303 and is passed via register a1
    Compute 3rd argument is 404 and is passed via register a2
    Return value is 99 and is returned via register a0
    The entry point of the "compute" function is at address 0x4200bbcc
    The return instruction for the "compute" function is at address 0x4200bcde
```

Justify how you obtained each value, including which GDB commands were used and what you observed.

Let me know if you need further clarification on any of these steps or if you encounter specific challenges while working on this part of the lab.

I have a raspberry pi set up to connect to the ESP32 and run everything, but it's relatively slow and clunky. Can I use WSL or my Mac to do lab 2 natively?

Yes, you can use either WSL (Windows Subsystem for Linux) or your Mac to complete Lab 2, but there are some considerations for each option:

1. Using WSL (Windows Subsystem for Linux)

- **ESP-IDF Compatibility**: WSL 2 works quite well with the ESP-IDF toolchain. You can install ESP-IDF directly in WSL, and it's generally faster and more reliable compared to WSL 1.
- **USB Device Access**: One challenge is accessing USB devices like the ESP32 from WSL. To make this work:
    - Ensure your version of Windows and WSL is up to date.
    - Use tools like Usbipd-Win to expose the USB port to WSL 2. This allows your ESP32 board to be detected within WSL.

- **Steps**:
   1. Set up WSL 2 if you haven't already.
   2. Install ESP-IDF and its dependencies as per the ESP32 documentation.
   3. Use `usbipd` to attach the ESP32 device to WSL, allowing OpenOCD and GDB to communicate with the board.

2. Using macOS

- **ESP-IDF Toolchain**: The ESP-IDF toolchain is officially supported on macOS, making it a good platform to use for ESP32 development.
- **Steps**:
   1. **Install Homebrew** (if not already installed): This will make installing dependencies easier.

```bash
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

   2. **Install ESP-IDF**: Follow the official ESP-IDF installation guide.

   - Use Homebrew to install required tools such as Python, Git, and CMake.
   - Run the ESP-IDF installation script to download and set up the toolchain.

   3. **USB Connection**: macOS should natively recognize the ESP32 device over USB, which allows OpenOCD to interface with it without the additional steps required for WSL.

Recommended Option

Using your **Mac** is probably the smoother and more straightforward option for Lab 2. The ESP-IDF setup on macOS is well-supported, and you can avoid the additional configuration overhead of USB forwarding that comes with WSL.

However, if you prefer working in a Linux-like environment on Windows, **WSL 2** with USB forwarding will also work, albeit with a bit more setup.

Let me know if you need specific guidance on setting up the ESP-IDF or running any of these tools on either platform!

So in the lab 2 document, it says that I need to figure out the following:
 - How can I upload an image obtained from someone else (where I do not have the source code)?
 - How can I use gdb against a binary/image?
Where do I start with this? I was able to disassemble the binary file into assembly so I have that ready as well.

Great job getting the binary disassembled into assembly! That's a solid first step. Let's address each question, breaking them down into actionable steps:

1. Uploading an Image Obtained from Someone Else

Since you don't have the source code, you need to work with the provided binary file (`lab2_debug.elf`). Here's how to handle this:

1. **Convert the ELF File to BIN**:
    - You have the `.elf` file, which is the RISC-V executable. You need to convert it to a format that can be flashed onto your ESP32 board. This format is usually `.bin`.
    - You can use `objcopy` to convert the ELF to a binary file:

    ```bash
    riscv64-unknown-elf-objcopy -O binary lab2_debug.elf lab2_debug.bin
    ```

2. **Flash the Binary to the ESP32**:
    - To upload the binary to the ESP32, you need to use a flashing tool. The ESP-IDF toolchain provides `idf.py flash`, but be careful with this command, as it might try to rebuild the project (potentially replacing the existing `.elf` file).
    - Instead, you can use `esptool.py` directly to flash the `.bin` file. This tool is versatile and used for flashing ESP32 devices:

    ```bash
    esptool.py --chip esp32c3 --port /dev/ttyUSB0 --baud 460800 write_flash 0x1000 lab2_debug.bin
    ```

        - Replace `/dev/ttyUSB0` with the port where your ESP32 is connected.
        - Make sure the ESP32 is in **bootloader mode** (typically by holding the BOOT button while pressing RESET, then releasing BOOT).

3. **OpenOCD for Debugging**:
    - Once the binary is flashed, you need to set up OpenOCD to enable GDB to communicate with the board.
    - Start OpenOCD using:

      ```bash
      idf.py openocd
      ```

        - This sets up a GDB server that allows you to connect to the ESP32 remotely.

2. Using GDB Against a Binary/Image

Now that you have the image on the board and OpenOCD is running, you can use GDB to debug the binary. The ELF file still serves an important purpose because it includes symbol information that helps in debugging, even if it's just minimal.

1. **Launch GDB with the ELF File**:
    - Start GDB with the ELF file and connect it to OpenOCD:

      ```bash
      riscv-esp32-elf-gdb -x gdbinit lab2_debug.elf
      ```

        - The `-x gdbinit` argument tells GDB to use the initialization file to set up the initial state, connect to OpenOCD, and set any necessary breakpoints or options.
2. **Configure GDB to Connect to the Board**:
    - In the `gdbinit` file, you have the following commands:

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

        - `target remote :3333` connects GDB to OpenOCD running on port `3333`.

    - `mon reset halt` resets the board and halts it at the entry point. You may need to use this frequently if the connection is lost.
    - `b app_main` sets a breakpoint at the `app_main` function, which is typically the main entry point in ESP32 applications. Since the binary was compiled without debug symbols, you may need to set breakpoints using specific addresses.
3. **Setting Breakpoints and Stepping Through Code**:
    - Use `objdump` to disassemble the ELF file and identify key locations (e.g., the `compute` function):

      ```bash
      riscv64-unknown-elf-objdump -D lab2_debug.elf > lab2_debug.S
      ```

    - In GDB, set breakpoints using addresses obtained from the disassembly. For example, if you see that the `compute` function starts at address `0x4200bbcc`:

      ```gdb
      b *0x4200bbcc
      ```

        - You can also set breakpoints at specific return instructions by specifying their addresses (e.g., `b *0x4200bcde`).
4. **Analyze Registers for Function Parameters and Return Value**:
    - The RISC-V calling convention typically passes function arguments in registers `a0`, `a1`, `a2`, and returns a value in `a0`.
    - Once you hit a breakpoint at the entry point of `compute`, inspect the relevant registers:

      ```gdb
      info registers
      ```

        - This command will display the values in all the registers. From there, check `a0`, `a1`, and `a2` to identify the parameters.
    - After stepping through the function, inspect `a0` again to find the return value.
5. **Navigate the Code Using GDB Commands**:
    - **Step through instructions**:

  - Use `stepi` to execute one assembly instruction at a time. This is particularly helpful for tracing how values move through registers and for understanding control flow.

- **View disassembled code**:

  ```gdb
  disassemble compute
  ```

  - This shows the disassembled instructions of the `compute` function, allowing you to follow along with how the arguments are used and where the return occurs.

6. **Handling GDB Instability**:

   - If GDB becomes unstable (e.g., with "Connect failed" errors), you may need to:
     - Unplug and re-plug the ESP32 board.
     - Re-run `mon reset halt` in GDB.
     - Restart GDB or OpenOCD as necessary.
   - Avoid pressing the hardware reset button; instead, use `mon reset halt` from GDB to reset the board in a controlled way.

Summary

1. **Upload the Image**:
   - Convert ELF to BIN, then use `esptool.py` to flash it to the ESP32.
   - Use OpenOCD to set up communication between the board and GDB.
2. **Debug with GDB**:
   - Start GDB with the ELF file and connect it to OpenOCD.
   - Set breakpoints based on the disassembled code.
   - Use GDB commands (`info registers`, `stepi`, `disassemble`) to analyze the function's arguments and return value.

You're off to a great start since you have the disassembly ready. Focus on converting the ELF to BIN, flashing it, and setting up OpenOCD and GDB to debug. Let me know if you have more questions about specific commands or run into any issues during these steps!

↳ Replace `/dev/ttyUSB0` with the port where your ESP32 is connected.

How do I determine the port which the ESP32 is connected to? Using usbipd, I know it's connected to port 1-10. Does that mean I just do ttyUSB10?

The port mapping shown by `usbipd` (e.g., `port 1-10`) actually refers to the Windows USB bus and port, not directly to the Linux device name in WSL.

To determine the correct serial port in WSL (or on Linux/macOS) for the ESP32, follow these steps:

Steps to Determine the Correct Port:

1. **Check the USB Port Name**:
   - After attaching the USB device with `usbipd`, you need to identify which `/dev/tty` device the ESP32 is using in WSL.
   - Run the following command to list all connected serial devices:

     ```bash
     dmesg | grep tty
     ```

     - This command will output the latest connected devices, including any new `/dev/ttyUSB` or `/dev/ttyACM` ports.
     - Look for something like `/dev/ttyUSB0`, `/dev/ttyUSB1`, or `/dev/ttyACM0`—this will be the port assigned to the ESP32.
2. **Verify the Port**:
   - Once you identify the possible port (e.g., `/dev/ttyUSB0`), you can verify it further by:
     - Running:

       ```bash
       ls /dev/ttyUSB*
       ```

       - This will list all the `/dev/ttyUSB` devices currently available. If your ESP32 is correctly connected, you should see a port like `/dev/ttyUSB0` or another number.
3. **Use the Port in Your Command**:
   - Now that you have identified the correct serial port, you can use it in your flashing command:

```bash
esptool.py --chip esp32c3 --port /dev/ttyUSB0 --baud 460800 write_flash
0x1000 lab2_debug.bin
```

- Replace `/dev/ttyUSB0` with the actual port name if it's different.

Notes:

- If there are multiple USB serial devices connected, it might be `/dev/ttyUSB1`, `/dev/ttyUSB2`, etc., so make sure to double-check the output from `dmesg`.
- On macOS, the port is typically named something like `/dev/cu.SLAB_USBtoUART` or `/dev/tty.SLAB_USBtoUART` for devices connected through USB-to-serial converters.

If you're still unsure which port to use, let me know, and we can further explore troubleshooting!

↳ To upload the binary to the ESP32, you need to use a flashing tool. The ESP-IDF toolchain provides `idf.py flash`, but be careful with this command, as it might try to rebuild the project (potentially replacing the existing `.elf` file).

I've converted the ELF file to binary. What do I do to be able to run idf.py flash? I need at least a CmakeLists file

To be able to use `idf.py flash` with the binary you have obtained, you need a minimal ESP-IDF project setup, which includes at least a `CMakeLists.txt` file and some project structure. Here's how you can do that:

Step-by-Step Guide to Prepare `idf.py flash`

1. **Create the Directory Structure**
   - Start by creating a project directory that includes the following subdirectories and files:

```plaintext
my_flash_project/
├── CMakeLists.txt
├── sdkconfig
└── main/
    ├── CMakeLists.txt
    └── flash_bin.c
```

- `my_flash_project/` is the root of your ESP-IDF project.
- `sdkconfig` will hold the default configuration settings.
- `main/` directory is where the code and related CMake configuration will be.

2. **Create `CMakeLists.txt` at the Project Root**
   - This is the main CMake file that defines the project:

```cmake
cmake_minimum_required(VERSION 3.16)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(flash_bin_project)
```

   - This tells ESP-IDF where to find the tools it needs to configure and build the project.

3. **Create `main/CMakeLists.txt`**
   - Create the CMake file in the `main/` directory:

```cmake
idf_component_register(SRCS "flash_bin.c"
                       INCLUDE_DIRS ".")
```

   - This file tells CMake that you will be using `flash_bin.c` as a source file. However, this source file is just a placeholder since you're only interested in flashing the binary.

4. **Create a Minimal Source File: `main/flash_bin.c`**
   - You need at least one C file for the CMake setup to work properly. You can create a minimal `flash_bin.c` like this:

```c
void app_main(void) {
    // This is a placeholder function for the ESP-IDF build system.
}
```

   - This placeholder function is just here so that `idf.py` can build the project, but it won't be executed when you flash your custom binary.

5. **Add or Modify `sdkconfig`**
   - You can generate a default `sdkconfig` file by running:

```bash
idf.py menuconfig
```

- Alternatively, you can copy an existing `sdkconfig` file from another project or leave it empty for default configurations.

6. **Build the Project Without Overwriting Your ELF**

   - To make sure you don't overwrite the provided ELF file (`lab2_debug.elf`), build a minimal project to generate the necessary partition table and bootloader binaries:

   ```bash
   idf.py build
   ```

   - This command will generate all the necessary files needed to flash a project (bootloader, partition table, and a dummy application binary).

7. **Replace the Application Binary with Your Own Binary**

   - The output folder will contain a `build` directory with various binaries, including:
     - Bootloader (`bootloader.bin`)
     - Partition table (`partition-table.bin`)
     - Application binary (`flash_bin.bin` , in this case).
   - Replace the generated application binary with your converted binary (`lab2_debug.bin`) using the following command:

   ```bash
   cp lab2_debug.bin build/flash_bin.bin
   ```

   - This ensures that when you flash the application, it uses your precompiled binary rather than the placeholder one.

8. **Flash the Binary Using `idf.py flash`**

   - Now, use `idf.py flash` to flash the complete set of binaries to your ESP32:

   ```bash
   idf.py flash
   ```

- This command will flash the bootloader, partition table, and your custom binary (`lab2_debug.bin`) onto the ESP32.

Important Considerations

- **Custom Binary**: The process ensures that your binary (`lab2_debug.bin`) is used instead of the one generated during the default build. This is key to preventing the ELF file from being overwritten or replaced.
- **Flashing Addresses**: If your custom binary has a specific entry address or partition scheme, you may need to adjust the `partition_table.csv` file (in the root directory) and reconfigure the project accordingly.
- **idf.py Targets**: You can use `idf.py -p PORT flash` to specify the exact port if you know the USB port assigned to your ESP32, e.g., `/dev/ttyUSB0`.

This setup allows you to utilize `idf.py` to flash your custom binary in a standard ESP-IDF project format, ensuring that the ESP32 is correctly initialized and flashed without unnecessary overwriting or re-compiling the provided ELF file.

idf_py_stdout_output_820409
File

This is what happens when I try to run idf.py flash after following the steps above

It looks like you uploaded an output file from running `idf.py flash`. Unfortunately, I can't directly view the file you uploaded since it isn't accessible to me.

To help you more effectively, could you copy and paste the error message or the key parts of the output that show what went wrong? Common issues with `idf.py flash` include:

- **Missing toolchain components**.
- **Incorrectly configured** `CMakeLists.txt`.
- **Errors related to USB connection**.

If you provide more information, I can guide you to a solution more specifically!

[1/5] cd /home/ssidor/esp/lab2/build/esp-idf/esptool_py && /home/ssidor/.espressif/python_env/idf5.5_py3.12_env/bin/python /home/ssidor/esp/esp-idf/components/partition_table/check_sizes.py --offset

0x8000 partition --type app
/home/ssidor/esp/lab2/build/partition_table/partition-table.bin /home/ssidor/esp/lab2/build/lab2.bin
FAILED: esp-idf/esptool_py/CMakeFiles/app_check_size
/home/ssidor/esp/lab2/build/esp-idf/esptool_py/CMakeFiles/app_check_size
cd /home/ssidor/esp/lab2/build/esp-idf/esptool_py &&
/home/ssidor/.espressif/python_env/idf5.5_py3.12_env/bin/python /home/ssidor/esp/esp-idf/components/partition_table/check_sizes.py --offset 0x8000 partition --type app
/home/ssidor/esp/lab2/build/partition_table/partition-table.bin /home/ssidor/esp/lab2/build/lab2.bin
Error: app partition is too small for binary lab2.bin size 0x5ffb36e:
 - Part 'factory' 0/0 @ 0x10000 size 0x100000 (overflow 0x5efb36e)
[2/5] Performing build step for 'bootloader'
[1/1] cd /home/ssidor/esp/lab2/build/bootloader/esp-idf/esptool_py &&
/home/ssidor/.espressif/python_env/idf5.5_py3.12_env/bin/python /home/ssidor/esp/esp-idf/components/partition_table/check_sizes.py --offset 0x8000 bootloader 0x0
/home/ssidor/esp/lab2/build/bootloader/bootloader.bin
Bootloader binary size 0x51c0 bytes. 0x2e40 bytes (36%) free.
ninja: build stopped: subcommand failed.

The error you're encountering indicates that the size of the `lab2.bin` binary file is too large for the allocated partition on the ESP32. Specifically, it says:

```arduino
Error: app partition is too small for binary lab2.bin size 0x5ffb36e:
  - Part 'factory' 0/0 @ 0x10000 size 0x100000 (overflow 0x5efb36e)
```

This means that the partition currently defined in your partition table is not large enough to accommodate the size of your binary.

Steps to Resolve the Partition Size Issue

1. **Modify Partition Table**:

- The default partition size for the application (`factory`) is likely too small to accommodate the large binary file you have (`lab2.bin`).
- You need to edit the partition table to create a larger partition for the application.
- In your project directory, locate the partition table CSV file. It should be named something like `partition_table.csv` , and if you don't have one, you can create it in the root of your project. Use the following content as an example:

```csv
# Name,   Type, SubType, Offset,   Size
nvs,      data, nvs,     0x9000,   0x5000
otadata,  data, ota,     0xe000,   0x2000
app0,     app,  factory, 0x10000,  0x600000   # Increase this size to accommodate your binary
```

  - The important line here is `app0` which sets the size of the factory application partition. The default might be `0x100000` (1MB), but your binary is significantly larger, so you need to increase it. You could try setting it to `0x600000` (6MB) or larger.

2. **Update** `CMakeLists.txt` :
   - You need to tell the ESP-IDF build system to use your updated partition table. Modify the `CMakeLists.txt` file in the root of your project to include:

```cmake
set(PARTITION_TABLE_CSV "${CMAKE_SOURCE_DIR}/partition_table.csv")
```

3. **Rebuild the Project**:
   - Run the following command to clean the build and apply the updated partition table:

```bash
idf.py fullclean
```

   - Then build the project again:

```bash
```

```
idf.py build
```

4. **Flash the Updated Application**:

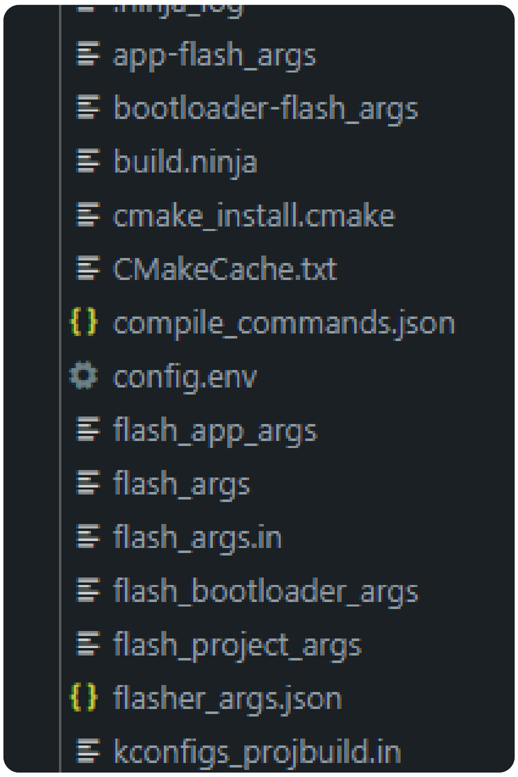
   - Now, with the updated partition table, the partition should be large enough to accommodate your binary:

     ```bash
     idf.py flash
     ```

Notes:

- **Partition Sizes**: Make sure the new partition sizes fit within the total flash memory of your ESP32 chip. Most ESP32 chips have a total of 4MB or 8MB of flash, so adjust the partition sizes accordingly.
- **Binary Size**: If your binary file is extremely large (larger than 4MB), it might indicate a misconfiguration or an unintentional inclusion of unnecessary components during the compilation.

This should help resolve the "app partition is too small" error you're encountering. Let me know if you need more help with this or if another error pops up.



This is what I have to work with. I can't find a .csv file

Thanks for sharing the screenshot! It looks like the partition table CSV file isn't present by default in your project. You can create a new partition table CSV file and tell CMake to use it.

Step-by-Step to Add a Partition Table CSV

1. **Create a Partition Table CSV File**:
   - In your project root directory (e.g., `/home/ssidor/esp/lab2/`), create a new file named `partition_table.csv` with the following content:

     ```csv
     # Name,   Type, SubType, Offset,   Size
     nvs,      data, nvs,     0x9000,   0x5000
     otadata,  data, ota,     0xe000,   0x2000
     app0,     app,  factory, 0x10000, 0x600000  # Increased the size of the application partition
     ```

     - The key line here is the `app0` line, which sets the size of the application partition. You can adjust this size to fit your binary. Since your binary is very large, I've increased the partition to 6MB (`0x600000`).
2. **Modify the `CMakeLists.txt` to Use the New Partition Table**:
   - In your project root directory (`/home/ssidor/esp/lab2/`), open or edit the `CMakeLists.txt` file to include the new partition table.
   - Add the following line before the `project()` command to tell the build system to use your custom partition table:

     ```cmake
     set(PARTITION_TABLE_CSV "${CMAKE_SOURCE_DIR}/partition_table.csv")
     ```
3. **Rebuild the Project**:
   - Clean the build to remove any cached settings that might interfere:

     ```bash
     idf.py fullclean
     ```
   - Now build the project again:

     ```bash
     ```

```
idf.py build
```

4. **Flash the Updated Application**:

   - Once the build completes successfully, you can flash the binary:

     ```bash
     idf.py flash
     ```

Summary:

- **Create** `partition_table.csv`: Add it to your project root directory to define custom partition sizes.
- **Update** `CMakeLists.txt`: Include a reference to use the new partition table.
- **Clean and Rebuild**: Run `idf.py fullclean` followed by `idf.py build`.
- **Flash**: Use `idf.py flash` to flash the updated application.

This should allow your larger binary to fit into the partition without triggering the "app partition is too small" error. Let me know if you run into any further issues!

Once I fullclean and build, do I have to replace the application binary again?

Yes, after running `idf.py fullclean` and then `idf.py build`, you will need to replace the generated application binary (`flash_bin.bin` or equivalent) again with your own binary (`lab2.bin`), as the `fullclean` will delete the previous build artifacts, including the `flash_bin.bin` that you replaced earlier.

Steps to Replace the Application Binary Again:

1. **Run** `idf.py fullclean`:

   - This command will remove all build artifacts, including the binary file that you had replaced previously.

2. **Rebuild the Project**:

   - After cleaning, run:

     ```bash
     ```

```
idf.py build
```

- This will generate all required files, including the partition table, bootloader, and a new application binary.

3. **Replace the Generated Application Binary**:

   - After the build completes, locate the newly generated application binary in the `build` directory.
   - Replace it with your `lab2.bin` using the following command:

     ```bash
     cp /path/to/lab2.bin /home/ssidor/esp/lab2/build/flash_bin.bin
     ```

   - Make sure to replace `/path/to/lab2.bin` with the actual path to your custom binary, and adjust the filename if the generated binary has a different name.

4. **Flash the Updated Application**:

   - Now, run `idf.py flash` to flash the updated binary along with the rest of the components (bootloader, partition table, etc.) to the ESP32:

     ```bash
     idf.py flash
     ```

Important Notes:

- You need to replace the generated application binary every time you perform a clean and rebuild operation.
- Ensure that the filename matches the one used during the build process (e.g., `flash_bin.bin` or whatever the build system generated). This guarantees that the correct binary is used during flashing.

Following these steps ensures that your custom binary is correctly flashed to the ESP32 after making any changes to the project setup.

```
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 002: ID 303a:1001 Espressif USB JTAG/serial debug unit
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
```

I need the bus number and port number from this image

From the information shown in the image, you can identify the **bus number** and **device number** of the connected ESP32 device.

In the output you provided:

```bash
Bus 001 Device 002: ID 303a:1001 Espressif USB JTAG/serial debug unit
```

- **Bus Number**: `001`
- **Device Number**: `002`

These identifiers (`Bus 001` and `Device 002`) can be used with tools like `usbipd` to identify and attach the USB device. However, they are not directly related to the Linux `/dev/ttyUSB*` port name you need for flashing.

How to Get the `/dev/ttyUSB*` Port

To identify which `/dev/ttyUSB*` or `/dev/ttyACM*` port is assigned to your ESP32:

1. **List Serial Devices**:
   - Run the following command to see the current serial devices:
     ```bash
     ls /dev/ttyUSB* /dev/ttyACM*
     ```
2. **Use `dmesg` for More Information**:
   - After plugging in your ESP32 device, use:
     ```bash
     dmesg | grep tty
     ```
   - This command will show recent messages and should include information about the USB serial device being attached, such as:
     ```arduino
     [12345.678901] usb 1-1: cp210x converter now attached to ttyUSB0
     ```
   - In this example, the ESP32 is connected to `/dev/ttyUSB0`.
3. **Confirm the Port**:

- Use a tool like `minicom` or `screen` to test the port, if needed:

  ```bash
  screen /dev/ttyUSB0 115200
  ```

- This command will open a serial console on the device. If you see output when resetting the ESP32, you know it's the correct port.

Once you have identified the correct port (e.g., `/dev/ttyUSB0`), you can use it in commands like `esptool.py` or `idf.py flash` to communicate with the ESP32:

```bash
idf.py -p /dev/ttyUSB0 flash
```

Let me know if you need more help with this, or if you encounter any issues identifying the correct port!

Does using c (continue) in gdb allow the program to run until the next breakpoint?

Yes, the `c` (or `continue`) command in GDB allows the program to run from its current position until it reaches the next breakpoint, a watchpoint, or another type of stop condition, such as a signal.

Here's how `c` works in GDB:

- **Command**: `c` or `continue`
- **Behavior**: The program resumes execution and will continue running until it:
  - Encounters the next **breakpoint** that you've set.
  - Hits a **watchpoint**, which is a special kind of breakpoint that pauses execution when the value of an expression changes.
  - Receives a **signal** that halts the execution, which could be an error, an exception, or a manual interrupt (e.g., pressing `Ctrl+C`).
  - Reaches the **end of the program**.

If no breakpoints are set, `c` will allow the program to run until it naturally completes or crashes.

This command is especially useful when you have set multiple breakpoints and want to skip over long sections of code, continuing execution until the program reaches a point of interest.

Example Scenario

Let's say you have set two breakpoints:

- **Breakpoint 1** at function `compute`
- **Breakpoint 2** at function `main_loop`

If you start running the program and use `c` after hitting **Breakpoint 1**, GDB will continue executing the code until it hits **Breakpoint 2**.

If you haven't set another breakpoint and use `c`, the program will run to completion or until it encounters an error.

Related GDB Commands

- `step` **(or** `s`**)**: Executes the next line of code and, if the line is a function call, steps into the function.
- `next` **(or** `n`**)**: Executes the next line of code, but steps over function calls.
- `finish`: Runs until the current function is finished.
- `until`: Continues running until it reaches a line greater than the current one or reaches a specified line.

Summary

Using `c` allows you to efficiently navigate through your program by running it until the next breakpoint or a stopping condition, which is helpful when you have already identified points of interest and want to avoid stepping through every line of code.

```
t0             0x4005890e      1074104590
t1             0x30000000      805306368
t2             0x0      0
fp             0x3fc9a080      0x3fc9a080
s1             0x3c0234cc      1006777548
a0             0x25     37
a1             0x0      0
a2             0x4      4
a3             0x0      0
a4             0x21     33
a5             0x25     37
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
t5             0x0      0
```

I'm currently running GDB on the .elf file. I need to figure out how to set a breakpoint at the beginning of the compute command, because I need to know what arguments are being passed into it by register value. I disassembled the binary file and found the exact addresses at which compute() is executing, but I keep getting this function not defined problem that's not letting me set a breakpoint. What do I do?

It looks like you're having trouble setting a breakpoint directly in GDB due to a "function not defined" issue, likely because GDB cannot symbolically recognize the address as belonging to the `compute` function. This is common when dealing with binaries without debug symbols.

Since you've disassembled the binary and found the addresses where the `compute()` function executes, you can set a breakpoint directly by using the **address of the function**. Here's how to proceed:

Steps to Set a Breakpoint by Address

1. **Set a Breakpoint at the Start of the `compute` Function**:
    - You mentioned that you have the addresses for where the `compute()` function begins. You can use the `b *address` syntax in GDB to set a breakpoint at a specific address. For example, if the start address of `compute` is `0x4200bba6`:

    ```gdb
    b *0x4200bba6
    ```

- The `*` before the address tells GDB that you are setting a breakpoint by address rather than by function name.

2. **Verify Register Values**:
   - Once the program hits the breakpoint at the beginning of `compute()`, you can inspect the **register values** to determine the arguments being passed.
   - RISC-V uses the following registers for function arguments:
     - `a0` **to** `a7`: These registers are used to pass arguments to functions.
   - To check the values of these registers, use:

     ```gdb
     info registers
     ```

   - This will show the current values of all registers, including `a0`, `a1`, `a2`, etc., which should contain the arguments for the `compute()` function.

3. **Disassemble the `compute` Function to Understand Its Flow**:
   - You can also disassemble the function to understand more about what it's doing and where you may need additional breakpoints.
   - Use:

     ```gdb
     disassemble 0x4200bba6
     ```

   - This will give you a listing of the assembly instructions for `compute()`, starting from the given address.

4. **Handling the Pending Breakpoint Prompt**:
   - When setting a breakpoint at an address that is part of a shared library or whose symbol is not yet loaded, GDB may ask if you want to make the breakpoint **pending**. This usually happens when the function has not yet been loaded.
   - You can avoid this by running the program up to a point where the function is loaded, or you can simply type `y` to allow GDB to set a pending breakpoint, which will activate once the code at that address is loaded.

Example Commands:

- **Set Breakpoint by Address**:

```gdb
b *0x4200bba6
```

- **Run/Continue Execution**:

```gdb
c
```

- **Check Register Values When Breakpoint is Hit**:

```gdb
info registers
```

- **Disassemble from the Address**:

```gdb
disassemble 0x4200bba6
```

Summary:

- Since you don't have symbolic information for `compute`, set the breakpoint by **address**.
- Use `b *address` to set the breakpoint, where `address` is the start address of `compute`.
- Once the breakpoint hits, inspect the argument registers (`a0`, `a1`, `a2`, etc.) using `info registers`.

This should allow you to determine the argument values being passed into `compute()` based on register contents. If you run into any further issues, feel free to share more details!

```
4200bb7e <compute>:
4200bb7e: 1101        addi   sp,sp,-32
4200bb80: ce22        sw  s0,28(sp)
4200bb82: 1000        addi   s0,sp,32
4200bb84: fea42623    sw  a0,-20(s0)
4200bb88: feb42423    sw  a1,-24(s0)
4200bb8c: fec42223    sw  a2,-28(s0)
4200bb90: fec42703    lw  a4,-20(s0)
4200bb94: fe842783    lw  a5,-24(s0)
```

4200bb98: 973e add a4,a4,a5
4200bb9a: fe442783 lw a5,-28(s0)
4200bb9e: 97ba add a5,a5,a4
4200bba0: 853e mv a0,a5
4200bba2: 4472 lw s0,28(sp)
4200bba4: 6105 addi sp,sp,32
4200bba6: 8082 ret

Can you please interpret this assembly? RISC-V

Sure, I'd be happy to help you interpret this RISC-V assembly code! I'll break it down instruction by instruction.

Here's the disassembled `compute` function:

```yaml
4200bb7e <compute>:
4200bb7e:   1101                 addi    sp, sp, -32
4200bb80:   ce22                 sw      s0, 28(sp)
4200bb82:   1000                 addi    s0, sp, 32
4200bb84:   fea42623             sw      a0, -20(s0)
4200bb88:   feb42423             sw      a1, -24(s0)
4200bb8c:   fec42223             sw      a2, -28(s0)
4200bb90:   fec42703             lw      a4, -20(s0)
4200bb94:   fe842783             lw      a5, -24(s0)
4200bb98:   973e                 add     a4, a4, a5
4200bb9a:   fe442783             lw      a5, -28(s0)
4200bb9e:   97ba                 add     a5, a5, a4
4200bba0:   853e                 mv      a0, a5
4200bba2:   4472                 lw      s0, 28(sp)
4200bba4:   6105                 addi    sp, sp, 32
4200bba6:   8082                 ret
```

Let's walk through the function step by step:

Step-by-Step Explanation

1. **Function Prologue: Setting Up the Stack Frame**

```yaml
4200bb7e:   1101                 addi    sp, sp, -32
```

- `addi sp, sp, -32` : This decreases the stack pointer (`sp`) by 32 bytes, creating space on the stack for storing local variables and saved registers. This sets up a stack frame for the function.

```makefile
4200bb80:   ce22                    sw      s0, 28(sp)
```

- `sw s0, 28(sp)` : Saves the value of register `s0` into the stack at an offset of 28 bytes from the stack pointer. This is done because `s0` will be used to keep track of the frame pointer, and it must be restored later.

```yaml
4200bb82:   1000                    addi    s0, sp, 32
```

- `addi s0, sp, 32` : Sets `s0` as the frame pointer for the current function. This allows consistent access to local variables regardless of further changes to `sp` . Here, `s0` points to the stack space created for this function.

2. **Storing Function Arguments**

```css
4200bb84:   fea42623             sw      a0, -20(s0)
```

- `sw a0, -20(s0)` : Stores the first argument (`a0`) to the stack at offset `-20` from `s0` . This creates a local variable that can be used throughout the function.

```css
4200bb88:   feb42423             sw      a1, -24(s0)
```

- `sw a1, -24(s0)` : Stores the second argument (`a1`) to the stack at offset `-24` from `s0` .

```css
4200bb8c:   fec42223             sw      a2, -28(s0)
```

- `sw a2, -28(s0)`: Stores the third argument (`a2`) to the stack at offset `-28` from `s0`.

3. **Loading Arguments and Performing Arithmetic**

```css
4200bb90:   fec42703            lw      a4, -20(s0)
```

- `lw a4, -20(s0)`: Loads the value of the first argument (previously stored at `-20(s0)`) into register `a4`.

```css
4200bb94:   fe842783            lw      a5, -24(s0)
```

- `lw a5, -24(s0)`: Loads the value of the second argument (previously stored at `-24(s0)`) into register `a5`.

```csharp
4200bb98:   973e                add     a4, a4, a5
```

- `add a4, a4, a5`: Adds the value in `a4` (first argument) to the value in `a5` (second argument), storing the result back in `a4`.

```css
4200bb9a:   fe442783            lw      a5, -28(s0)
```

- `lw a5, -28(s0)`: Loads the value of the third argument (previously stored at `-28(s0)`) into register `a5`.

```csharp
4200bb9e:   97ba                add     a5, a5, a4
```

- `add a5, a5, a4`: Adds the value in `a5` (third argument) to the value in `a4` (which is the sum of the first and second arguments). This stores the result in `a5`.

4. **Return the Result**

```bash
4200bba0:   853e                    mv      a0, a5
```

- `mv a0, a5`: Moves the value in `a5` (the result of the arithmetic operations) to `a0`. In RISC-V, the return value is usually passed back in `a0`, so this sets up the return value.

5. **Function Epilogue: Restoring the Stack**

```yaml
4200bba2:   4472                    lw      s0, 28(sp)
```

- `lw s0, 28(sp)`: Restores the saved value of `s0` from the stack. This is important to revert the frame pointer to its original state before the function call.

```yaml
4200bba4:   6105                    addi    sp, sp, 32
```

- `addi sp, sp, 32`: Restores the stack pointer by adding `32` back to it, effectively "deallocating" the stack space that was reserved for this function.

```yaml
4200bba6:   8082                    ret
```

- `ret`: Returns control to the caller. The return value is already in `a0`.

Summary of What the Function Does

- **Arguments**: The function takes **three arguments** in `a0`, `a1`, and `a2`.
- **Operation**:
  1. It adds `a0` and `a1`, storing the result in a temporary register.
  2. Then, it adds the third argument (`a2`) to that sum.
  3. Finally, it stores the result in `a0` and returns it.
- **Result**: The result is the sum of all three arguments, returned in `a0`.

In simple terms, this function is equivalent to the following C code:

```c
int compute(int a, int b, int c) {
    return a + b + c;
}
```

This explanation should help you understand how the function operates at the assembly level, and also help you determine how the arguments (`a0` , `a1` , `a2`) are being passed in and used during computation.