HACKING RISC-V

CHAPTER 1: WHY RISC-V

Introduction

As technology evolves, the importance of understanding the underlying hardware architecture becomes increasingly crucial for those interested in hacking and security. RISC-V, an open-source instruction set architecture (ISA), has gained significant traction in the industry due to its flexibility, scalability, and openness. Unlike proprietary ISAs, RISC-V provides the freedom to modify and extend the architecture, making it an ideal platform for experimentation and innovation. In this chapter, we will explore why learning hacking for RISC-V is essential, particularly in the context of using the ESP32-C3-DevKitM-1 development board and PlatformIO with Visual Studio Code. These tools provide an easy and accessible setup that will allow you to focus on practical projects and hands-on experience.

The Rise of RISC-V

RISC-V is not just another ISA; it represents a paradigm shift in the world of computing. Born out of a desire to create a simple, clean, and extensible architecture, RISC-V has attracted a growing community of developers, researchers, and companies. The open nature of RISC-V allows anyone to implement and modify the ISA without paying royalties or licensing fees, fostering innovation and collaboration. As RISC-V continues to gain popularity, especially in embedded systems, IoT devices, and security-critical applications, the demand for skilled professionals who understand this architecture is on the rise. Learning to hack on RISC-V not only equips you with knowledge of a cutting-edge technology but also positions you at the forefront of a rapidly expanding field.

Why Hacking?

Hacking, in its truest sense, is about understanding systems at a fundamental level, finding creative solutions to problems, and pushing the boundaries of what is possible. In the context of RISC-V, hacking can involve anything from reverse engineering binaries to exploiting vulnerabilities in hardware or firmware. By learning to hack RISC-V, you gain insight into how modern computing systems work, how to identify and mitigate security risks, and how to leverage the unique features of RISC-V for your own projects. Whether you're interested in cybersecurity, embedded systems, or just enjoy the challenge of solving complex problems, hacking on RISC-V offers a rich and rewarding experience.

Why the ESP32-C3-DevKitM-1?

The ESP32-C3-DevKitM-1 development board is a powerful yet cost-effective platform that is perfect for hacking and experimentation. Based on the ESP32-C3, a low-power, single-core RISC-V microcontroller with integrated Wi-Fi and Bluetooth, this development board offers a wide range of features for IoT and embedded applications. Its RISC-V core makes it an ideal candidate for learning and hacking RISC-V, while its built-in peripherals provide ample opportunities for hands-on projects. Whether you're interested in building a custom IoT device, experimenting with low-level firmware, or exploring the security of wireless communications, the ESP32-C3-DevKitM-1 offers a versatile platform to bring your ideas to life.

Why PlatformIO and Visual Studio Code?

Setting up a development environment can often be a daunting task, especially for those new to embedded systems or hacking. PlatformIO, combined with Visual Studio Code, simplifies this process by providing a user-friendly interface and a powerful set of tools for developing, debugging, and deploying firmware. PlatformIO supports a wide range of microcontrollers, including the ESP32-C3, and integrates seamlessly with Visual Studio Code, one of the most popular code editors in the world. With PlatformIO, you can easily configure your projects, manage dependencies, and even upload your code to the ESP32-C3-DevKitM-1 with just a few clicks. This streamlined workflow allows you to focus on what matters most: learning and hacking RISC-V.

Practical Projects: The Key to Mastery

Theory alone is not enough to master hacking on RISC-V. To truly understand the architecture and develop your skills, you need to engage in practical projects that challenge you to apply what you've learned. The combination of the ESP32-C3-DevKitM-1 and PlatformIO provides the perfect environment for these projects. From writing custom firmware to exploiting vulnerabilities in existing systems, the hands-on experience you gain will deepen your understanding and prepare you for real-world challenges. Throughout this book, we will guide you through a series of projects that will not only teach you the fundamentals of RISC-V hacking but also inspire you to explore new ideas and create your own innovations.

Conclusion

Learning to hack RISC-V is not just about mastering a new ISA; it's about gaining the skills and knowledge needed to thrive in a world where open-source hardware is becoming increasingly important. The ESP32-C3-DevKitM-1 development board, combined with PlatformIO and Visual Studio Code, provides an accessible and powerful platform for your journey into RISC-V hacking. As you progress through this book, you will gain the confidence to tackle complex challenges, the creativity to develop innovative solutions, and the expertise to contribute to the growing RISC-V community. Welcome to the world of RISC-V hacking—your adventure begins here.

CHAPTER 2: SETUP ENVIRONMENT

Introduction

Before diving into the exciting world of RISC-V programming on the ESP32-C3, it's essential to have the right tools and setup. This chapter will guide you through purchasing the ESP32-C3-DevKitM-1 development board, setting up your development environment with Visual Studio Code (VS Code), installing PlatformIO and C++ plugins, and configuring OpenOCD for debugging. By the end of this chapter, you'll be ready to start coding and debugging your projects.

1. Purchase the ESP32-C3-DevKitM-1 Development Board and Accessories

The ESP32-C3-DevKitM-1 is a versatile and powerful development board that is perfect for exploring RISC-V architecture. This board is compact, feature-rich, and provides everything you need to get started with RISC-V on the ESP32-C3.

Purchase Links:

https://www.amazon.com/Espressif-ESP32-C3-DevKitM-1-Development-Board/dp/B08W2J9B8J

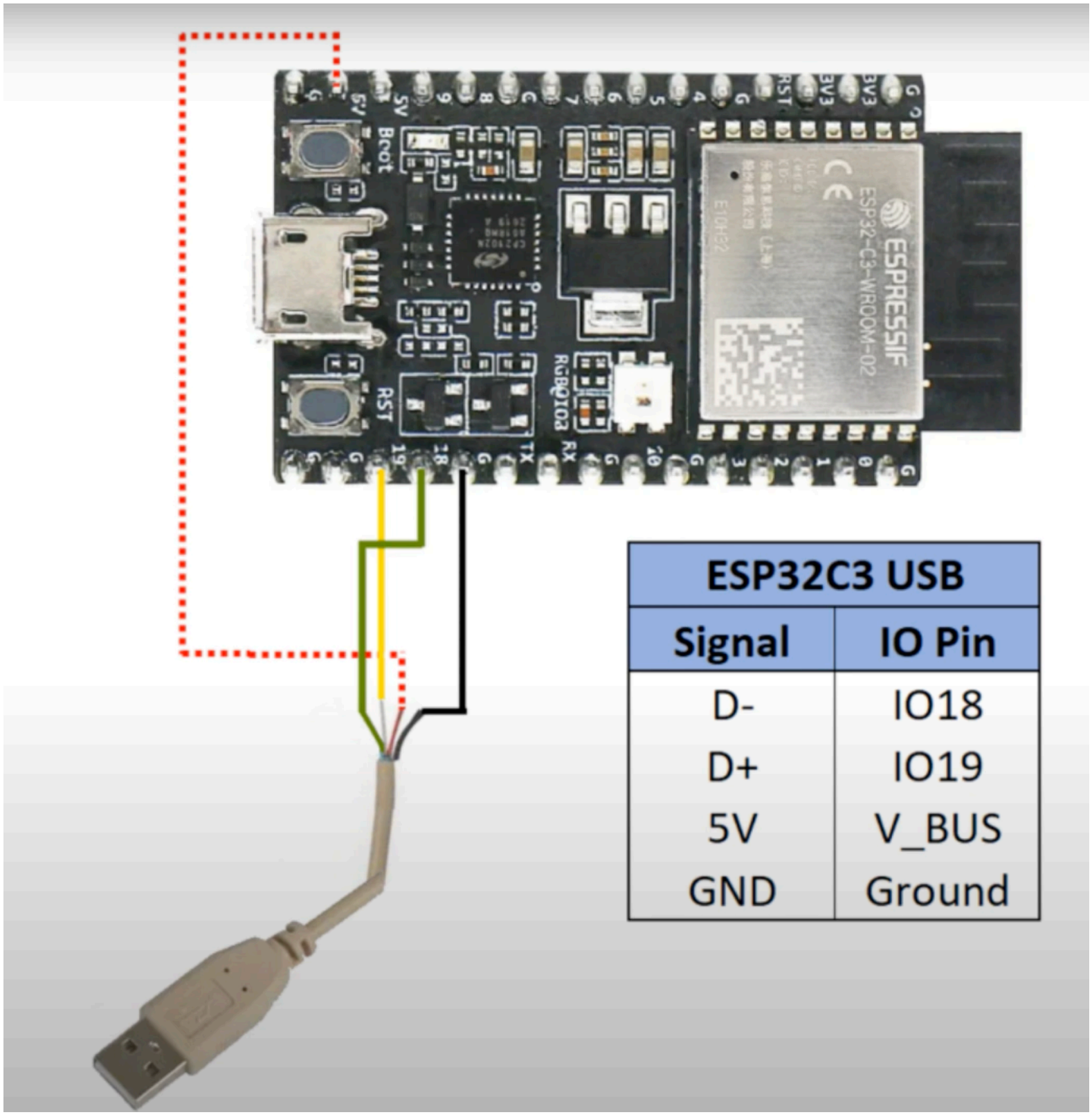
https://www.amazon.com/DGZZI-PL2303TA-Console-Serial-Raspberry/dp/B07W42V16T

https://www.amazon.com/Syntech-Adapter-Thunderbolt-Compatible-MacBook/dp/B07CVX3516

https://www.amazon.com/EL-CK-002-Electronic-Breadboard-Capacitor-Potentiometer/dp/B01ERP6WL4

2. Setup USB Cable

Follow the diagram to wire up your ESP32-C3 for flashing and debugging.



ESP32C3 USB	
Signal	IO Pin
D-	IO18
D+	IO19
5V	V_BUS
GND	Ground

2. Set Up Visual Studio Code (VS Code)

Visual Studio Code (VS Code) is a powerful and popular code editor that is widely used for embedded systems development. It supports multiple programming languages, including C++, and provides excellent tools for debugging and code management.

2.1 Download and Install VS Code

1. Visit the Visual Studio Code website - https://code.visualstudio.com.

2. Download the installer for your operating system (Windows, macOS, or Linux).

3. Follow the on-screen instructions to install VS Code.

2.2 Install the PlatformIO and C++ Plugins

PlatformIO is an open-source ecosystem that supports multiple development platforms, including the ESP32. It simplifies the process of writing, testing, and debugging embedded software.

1. Open VS Code.

2. Go to the Extensions view by clicking on the Extensions icon in the Activity Bar on the side of the window or by pressing `Ctrl+Shift+X`.

3. In the search box, type `PlatformIO IDE` and click "Install."

4. Next, search for `C/C++` and click "Install" to add the C++ extension, which provides language support for C++.

3. Install OpenOCD

OpenOCD (Open On-Chip Debugger) is essential for debugging your projects. Below are the installation steps for Windows, macOS, and Linux.

3.1 Windows

1. Download the latest release of OpenOCD for Windows from the [official website](http://openocd.org/getting-openocd).

2. Extract the downloaded ZIP file to a directory of your choice.

3. Add the OpenOCD *bin* directory to your system's *PATH* environment variable:

- Right-click *This PC* or *Computer*, and select *Properties*.

- Click on *Advanced system settings*, and then *Environment Variables*.

- Find the *Path* variable, click *Edit*, and add the path to the OpenOCD *bin* directory.

3.2 macOS

1. Open Terminal.

2. Install OpenOCD using Homebrew:

```
brew install openocd
```

3.3 Linux

1. Open Terminal.

2. Install OpenOCD using your package manager:

- Debian/Ubuntu:

```
sudo apt-get install openocd
```

- Fedora:

```
sudo dnf install openocd
```

- Arch Linux:

```
sudo pacman –S openocd
```

Conclusion

By following the steps outlined in this chapter, you will have all the necessary tools and configurations to start developing RISC-V projects on the ESP32-C3-DevKitM-1. With your development environment set up, you're ready to dive into the code and explore the capabilities of RISC-V on this powerful platform.

CHAPTER 3: HELLO, WORLD

Introduction

In this chapter, we will walk through creating your first "Hello World" project using PlatformIO with the ESP32-C3 and the Arduino framework. This project will demonstrate how to integrate an assembly function into a C++ project.

1. Open Visual Studio Code and Create a New PlatformIO Project

1.2 Create a New PlatformIO Project

1. Click on the PlatformIO icon in the left sidebar.

2. Click on the “New Project” button.

3. In the "Project Name" field, enter *0x01_hello_world*.

4. For "Board," select *Espressif ESP32-C3-DevKitM-1*.

5. For "Framework," select *Arduino*.

6. Choose a location to save the project and click “Finish.”

PlatformIO will now create a new project with the necessary folders and files.

2. Modify the *main.cpp* File

Once your project is created, you'll find a *main.cpp* file inside the *src* folder. Replace the contents of this file with the following code:

```
#include <Arduino.h>


// Declare the function implemented in assembly
extern "C" void hello_world();


void setup() {
    Serial.begin(115200);
    hello_world();  // Call the assembly function directly
}

void loop() {
}
```

This *main.cpp* file initializes the serial communication at a baud rate of 115200 and calls the *hello_world* function, which will be implemented in assembly language.

3. Create the Assembly Function

Next, you'll create a new file to hold your assembly code.

3.1 Create the *main.S* File

1. In the *src* folder, create a new file named *main.S*.

2. Add the following assembly code:

```
.global hello_world


hello_world:
    li      t0, 0
    ret
```

This simple assembly code defines a function named *hello_world* that does nothing and returns immediately.

4. Build and Upload the Project

With both the *main.cpp* and *main.S* files in place, you're ready to build and upload the project to your ESP32-C3 board.

1. Click on the checkmark icon in the bottom toolbar of VS Code to build the project.

2. If the build is successful, click on the right arrow icon to upload the code to your ESP32-C3 board.

5. Configure PlatformIO for the ESP32-C3

Once you have PlatformIO installed, you need to configure your project for the ESP32-C3. Below is a sample *platformio.ini* file that you can use for your projects:

```
[env:esp32-c3-devkitm-1]

platform = espressif32

board = esp32-c3-devkitm-1

framework = arduino

monitor_speed = 115200

debug_tool = esp-builtin

debug_init_break = break setup

build_type = debug
```

- **platform**: Specifies the platform you are working with, in this case, *espressif32*.

- **board**: The specific board you are using, which is *esp32-c3-devkitm-1*.

- **framework**: Indicates that you are using the Arduino framework.

- **monitor_speed**: Sets the serial monitor speed to *115200* baud.

- **debug_tool**: Specifies the built-in debugger for the ESP32-C3.

- **debug_init_break**: Sets an initial breakpoint at the `setup()` function.

- **build_type**: Configures the build for debugging.

Conclusion

Congratulations! You've successfully created a "Hello World" project that integrates a simple assembly function into your C++ code on the ESP32-C3 using the Arduino framework. This setup forms the foundation for more advanced projects that combine the power of C++ with the efficiency of assembly language.

CHAPTER 4: BINARY NUMBERS

Introduction

Binary numbers are the foundation of all digital systems, including the microcontrollers like the ESP32-C3 that you are working with in RISC-V assembly language. Understanding binary is crucial for manipulating data, controlling hardware, and performing low-level programming tasks. This chapter will guide you through the basics of binary numbers, how to calculate them by hand, and how they relate to the operation of microcontrollers.

1. Introduction to Binary Numbers

The binary number system is a base-2 numeral system that uses only two symbols: *0* and *1*. Each digit in a binary number is referred to as a "bit" (short for binary digit). In contrast to the decimal system, which is base-10 and uses digits *0* through *9*, the binary system only has two digits.

For example, the binary number *1011* represents the decimal number *11*. Each bit in the binary number has a place value that is a power of 2, starting from *2^0* on the right.

2. Binary Place Values

Just like the decimal system, where the place value of each digit increases by a power of 10 as you move from right to left, in binary, the place value increases by a power of 2. Here's how the place values work in binary:

- The rightmost bit (least significant bit, or LSB) is **2^0** (which equals 1).

- The next bit to the left is *2^1* (which equals 2).

- The next bit is *2^2* (which equals 4).

- The next bit is *2^3* (which equals 8).

- And so on...

For a binary number like *1101*, the place values are:

1 x 2^3 = 1 x 8 = 8

1 x 2^2 = 1 x 4 = 4

0 x 2^1 = 0 x 2 = 0

1 x 2^0 = 1 x 1 = 1

Adding these values together gives the decimal equivalent

8 + 4 + 0 + 1 = 13

Thus, *1101* in binary equals *13* in decimal.

3. Converting Decimal to Binary by Hand

To convert a decimal number to binary, you repeatedly divide the number by 2, keeping track of the quotient and the remainder. The binary number is formed by the remainders, starting from the least significant bit (the last remainder) to the most significant bit (the first remainder).

Let’s go through an example:

Convert Decimal *25* to Binary:

1. Divide 25 by 2:

   - Quotient = 12, Remainder = 1 → LSB

2. Divide 12 by 2:

   - Quotient = 6, Remainder = 0

3. Divide 6 by 2:

   - Quotient = 3, Remainder = 0

4. Divide 3 by 2:

   - Quotient = 1, Remainder = 1

5. Divide 1 by 2:

   - Quotient = 0, Remainder = 1 → MSB

Now, write the remainders in reverse order:

25 base 10 = 11001 base 2

So, the binary representation of decimal *25* is *11001*.

4. Binary Addition

Binary addition is similar to decimal addition, but it only involves the digits 0 and 1. The rules for binary addition are simple:

- 0 + 0 = 0

- 0 + 1 = 1

- 1 + 0 = 1

- 1 + 1 = 10 (which is 0 with a carry of 1 to the next higher bit)

Here’s an example of binary addition:

```
  1011  (11 in decimal)
+  110  (6 in decimal)
------
 10001  (17 in decimal)
```

5. Binary Subtraction

Binary subtraction uses borrowing, similar to decimal subtraction. The rules for binary subtraction are:

- 0 - 0 = 0

- 1 - 0 = 1

- 1 - 1 = 0

- 0 - 1 = 1 (with a borrow of 1 from the next higher bit)

Here’s an example of binary subtraction:

```
  1011  (11 in decimal)
-  110  (6 in decimal)
------
   101  (5 in decimal)
```

6. Bitwise Operations

Understanding binary is essential for performing bitwise operations, which are common in low-level programming. Bitwise operations manipulate individual bits of binary numbers. The most common bitwise operations are AND, OR, XOR, and NOT.

- **AND**: Compares each bit of two binary numbers. The result is *1* if both bits are *1*, otherwise *0*.

- **OR**: Compares each bit of two binary numbers. The result is *1* if at least one bit is *1*.

- **XOR** (Exclusive OR): Compares each bit of two binary numbers. The result is *1* if the bits are different.

- **NOT**: Inverts each bit of the binary number (0 becomes 1, and 1 becomes 0).

Example:

```
A =  1010 (10 in decimal)

B =  1100 (12 in decimal)


A AND B  =  1000 (8 in decimal)

A OR B   =  1110 (14 in decimal)

A XOR B  =  0110 (6 in decimal)

NOT A    =  0101 (5 in decimal)
```

7. Working with Larger Binary Numbers

As you work with microcontrollers like the ESP32-C3, you'll often encounter binary numbers that represent various settings or data, such as GPIO configurations, memory addresses, or data in registers.

For example, in a 32-bit system, the binary number could be much larger:

```
11010111001010010101001010101010
```

Converting this by hand would involve breaking it down into manageable sections, applying the place value method described earlier.

8. Applying Binary in RISC-V Assembly

Understanding binary is critical when writing assembly code, especially when setting or clearing specific bits in registers. For example, to enable a specific GPIO pin on the ESP32-C3, you might need to manipulate a specific bit in a register. Knowing how to convert between binary and hexadecimal (another common numeral system in programming) will also be valuable.

In RISC-V assembly, you'll often see instructions like:

```
li      t0, 0b00000001  # Load immediate binary 00000001 into
register t0

slli    t0, t0, 4     # Shift left logical by 4 bits (binary
00010000)
```

This example shows how binary is directly used in assembly language to manipulate data.

9. Conclusion

Understanding binary numbers is essential for anyone working with digital electronics or low-level programming, including assembly language on the ESP32-C3. By mastering binary calculations and conversions, you'll be well-equipped to write efficient, effective code and understand the inner workings of the hardware you're programming.

CHAPTER 5: HEXADECIMAL NUMBERS

Introduction

Hexadecimal (or hex) numbers are a base-16 numeral system commonly used in computing and digital electronics. They are an essential part of low-level programming, including assembly language, as they provide a more human-readable way to represent binary numbers. This chapter will explore the basics of hexadecimal numbers, how to convert between hex and other numeral systems, and their practical applications in programming the ESP32-C3 microcontroller.

1. Introduction to Hexadecimal Numbers

The hexadecimal system uses 16 symbols to represent values: the digits *0* to *9* and the letters *A* to *F*. Here's the complete list:

- 0 = 0

- 1 = 1

- 2 = 2

- 3 = 3

- 4 = 4

- 5 = 5

- 6 = 6

- 7 = 7

- 8 = 8

- 9 = 9

- A = 10

- B = 11

- C = 12

- D = 13

- E = 14

- F = 15

Each hex digit represents four binary digits (bits), making it easier to represent large binary numbers. For example, the binary number *1101* can be more compactly written as *D* in hex.

2. Converting Binary to Hexadecimal

Converting binary numbers to hexadecimal is straightforward. Since each hex digit corresponds to exactly four binary digits, you can group the binary number into sets of four, starting from the right. If the leftmost group has fewer than four digits, pad it with leading zeros.

Here's an example:

Convert Binary *110101101011* to Hexadecimal:

1. Group the binary digits into sets of four:

```
1101 0110 1011
```

2. Convert each group to its hex equivalent:

1101 & = D

0110 & = 6

1011 & = B

So, the binary number *110101101011* is *D6B* in hexadecimal.

3. Converting Hexadecimal to Binary

To convert a hexadecimal number to binary, simply reverse the process by converting each hex digit to its four-bit binary equivalent.

Convert Hexadecimal *3A7* to Binary:

1. Convert each hex digit to binary:

3 & = 0011

A & = 1010

7 & = 0111

`3A7 = 0011 1010 0111`

So, the hexadecimal number *3A7* is *001110100111* in binary.

4. Converting Decimal to Hexadecimal

To convert a decimal number to hexadecimal, repeatedly divide the number by 16, keeping track of the quotient and remainder. The remainders, read in reverse order, give you the hexadecimal number.

Let's go through an example:

Convert Decimal *956* to Hexadecimal:

1. Divide 956 by 16:

   - Quotient = 59, Remainder = 12 (*C* in hex)

2. Divide 59 by 16:

   - Quotient = 3, Remainder = 11 (*B* in hex)

3. Divide 3 by 16:

   - Quotient = 0, Remainder = 3

Now, write the remainders in reverse order:

956 base 10 = 3BC base 16

So, the decimal number *956* is *3BC* in hexadecimal.

5. Converting Hexadecimal to Decimal

To convert a hexadecimal number to decimal, multiply each digit by its place value and add the results.

Convert Hexadecimal *1F4* to Decimal:

1. Identify the place values (from right to left):

4 x 16^0 = 4 x 1 = 4

F x 16^1 = 15 x 16 = 240

1 x 16^2 = 1 x 256 = 256

2. Add the values together:

256 + 240 + 4 = 500

So, the hexadecimal number *1F4* is *500* in decimal.

6. Hexadecimal Arithmetic

Hexadecimal arithmetic follows the same principles as decimal and binary arithmetic, with carry-over rules adjusted for base-16. Here's a brief overview:

- **Addition**: If the sum of two hex digits exceeds *F* (15 in decimal), carry 1 to the next higher digit.

- **Subtraction**: If you subtract a larger hex digit from a smaller one, borrow 1 from the next higher digit.

Example: Add Hexadecimal *A3* and *1C*

```
  A3
+ 1C
----
  BF
```

Here, *3 + C* equals *F*, and *A + 1* equals *B*, so the result is *BF*.

7. Hexadecimal in RISC-V Assembly

In RISC-V assembly language, hexadecimal numbers are often used to represent memory addresses, data values, and instruction opcodes. Hexadecimal notation is more concise and readable compared to binary, making it easier to understand and write code.

For example, in assembly code, you might see:

```
li      t0, 0x1F4  # Load immediate hex 1F4 (500 in decimal) into register t0
```

This line loads the decimal value *500* into register *t0*, but it's written in hexadecimal as *0x1F4*.

8. Working with Hexadecimal and Memory

In microcontrollers like the ESP32-C3, memory addresses and register values are commonly represented in hexadecimal. Understanding how to read and manipulate these values is crucial for effective low-level programming.

For example, the base address of GPIO registers might be given as *0x60004000*, and you might need to manipulate specific bits within these registers to control hardware. Knowing how to convert and understand these hexadecimal values is essential.

9. Hexadecimal and Bitwise Operations

Just as with binary, hexadecimal numbers can be used in bitwise operations. These operations are often more convenient in hex because you can manipulate larger groups of bits at once.

Example:

```
A =  0xF0F0  # Binary: 1111000011110000

B =  0x0F0F  # Binary: 0000111100001111


A AND B  =  0x0000  # Binary: 0000000000000000

A OR B   =  0xFFFF  # Binary: 1111111111111111

A XOR B  =  0xFFFF  # Binary: 1111111111111111
```

In this example, using hexadecimal makes the bitwise operations clearer and more concise.

10. Conclusion

Hexadecimal numbers are an indispensable part of digital electronics and low-level programming. Their close relationship with binary numbers and their concise representation make them ideal for use in assembly language and understanding hardware interactions, such as with the ESP32-C3 microcontroller.

By mastering hexadecimal, you'll be better equipped to write efficient, clear code and understand the intricate details of how your programs interact with the hardware.

In the following chapter, we'll delve into how to work with hexadecimal in more advanced operations, including memory manipulation and addressing in RISC-V assembly language.

CHAPTER 6: ADVANCED OPS

Introduction

In this chapter, we will explore advanced operations using hexadecimal and binary, focusing on memory manipulation and bit-level operations. Understanding these concepts is crucial for programming the ESP32-C3 microcontroller in RISC-V assembly. We will cover bit manipulation, memory addressing, and how hexadecimal and binary representations are used to interact with hardware registers.

1. Introduction to Bit Manipulation

Bit manipulation involves the direct control of individual bits within a binary number. This is a fundamental technique in low-level programming, especially when dealing with hardware registers in microcontrollers like the ESP32-C3.

Each bit in a register can represent a specific function or control a particular aspect of the hardware. For example, a bit might enable or disable a GPIO pin, trigger an interrupt, or configure the mode of a peripheral.

1.1 Bitwise Operators in RISC-V

RISC-V assembly provides several bitwise operators that are essential for bit manipulation:

- **AND** (*and*): Clears specific bits by masking them.

- **OR** (*or*): Sets specific bits.

- **XOR** (*xor*): Toggles specific bits.

- **NOT** (via *xori* with *-1*): Inverts the bits.

- **Shift Left** (*sll*): Shifts bits to the left, multiplying the number by powers of two.

- **Shift Right b**: Shifts bits to the right, dividing the number by powers of two.

1.2 Example: Setting and Clearing Bits

Let's consider an example where you want to manipulate the GPIO0 register in the ESP32-C3 to control an LED. The GPIO registers are memory-mapped and can be manipulated using hexadecimal addresses.

Assume you need to set GPIO0 as an output by setting the corresponding bit in the *GPIO_ENABLE_REG*. The register might be located at address *0x60004020*, and you need to set the first bit (bit 0).

Set GPIO0 as Output:

1. Load the address of the GPIO_ENABLE_REG:

```
lui     t0, 0x60004         # Load upper immediate with base address

addi    t0, t0, 0x020       # Add the lower offset to reach
GPIO_ENABLE_REG
```

2. Set bit 0 in the register:

```
li      t1, 0x1             # Load immediate with bitmask 0000 0001

sw      t1, 0(t0)           # Store word t1 into the address pointed
by t0 (GPIO_ENABLE_REG)
```

This code sets bit 0 in the *GPIO_ENABLE_REG*, configuring GPIO0 as an output.

Clear GPIO0 (Disable Output):

To clear the bit, use the *AND* operation with the inverse of the bitmask:

```
lui     t0, 0x60004         # Load base address

addi    t0, t0, 0x020       # Offset to GPIO_ENABLE_REG

lw      t2, 0(t0)           # Load current register value into t2

li      t1, 0xFFFFFFFE      # Load bitmask with all 1s except bit 0
(1111 1111 1111 1111 1111 1111 1111 1110)

and     t2, t2, t1          # AND to clear bit 0

sw      t2, 0(t0)           # Store mod val to register
```

This clears bit 0, disabling GPIO0 as an output.

2. Memory Manipulation Using Hexadecimal

Memory manipulation is critical in microcontroller programming, as it allows you to control hardware peripherals, configure registers, and manage data. In RISC-V, memory addresses are often represented in hexadecimal for readability and conciseness.

2.1 Memory Addressing

Memory in the ESP32-C3 is organized in a linear address space. Specific addresses correspond to specific peripherals, and registers control these peripherals. Understanding how to manipulate these addresses is key to effective low-level programming.

For example, consider the base address for the GPIO registers on the ESP32-C3, *0x60004000*. To manipulate a specific GPIO pin, you need to calculate the exact address of the relevant register and perform bitwise operations on it.

Accessing a Register:

Let's access the *GPIO_OUT_REG* to set the state of a GPIO **pin:**

```
lui     t0, 0x60004         # Load upper part of base address
addi    t0, t0, 0x000       # Offset to GPIO_OUT_REG (if it's at
0x60004000)
```

With *t0* holding the base address, you can read or write to this register:

Write to GPIO_OUT_REG:

```
li      t1, 0x1             # Load immediate value to set GPIO0 high

sw      t1, 0(t0)           # Write to GPIO_OUT_REG to set GPIO0
```

This writes to the *GPIO_OUT_REG*, setting GPIO0 high.

2.2 Reading from a Register

To read a value from a register, load the address and then use the *lw* (load word) instruction:

```
lui     t0, 0x60004         # Load base address

addi    t0, t0, 0x000       # Offset to GPIO_OUT_REG

lw      t1, 0(t0)           # Load value from GPIO_OUT_REG into t1
```

After this, *t1* contains the current value of *GPIO_OUT_REG*, which can be examined or manipulated further.

3. Using Hexadecimal for Efficient Code

Hexadecimal notation is especially useful when dealing with memory-mapped registers, as it allows you to easily identify and manipulate specific bits. It also simplifies the code, making it more readable and less prone to errors.

3.1 Efficient Bit Masking

Bit masking is a technique used to isolate or modify specific bits within a register. By using hexadecimal, you can create masks that target specific bits with ease.

Example: Isolating a Single Bit:

To check if GPIO0 is set:

```
lui     t0, 0x60004         # Load base address
addi    t0, t0, 0x000       # Offset to GPIO_OUT_REG
lw      t1, 0(t0)           # Load current register value into t1
andi    t2, t1, 0x1         # AND t1 with mask 0x1 to isolate bit 0
```

If *t2* is non-zero, GPIO0 is set.

3.2 Combining Multiple Bits

You can also use hexadecimal to combine multiple bits. For instance, if you want to set both GPIO0 and GPIO1:

```
li      t1, 0x3             # Load 0x3 (binary 0000 0011) to set
GPIO0 and GPIO1
sw      t1, 0(t0)           # Write to GPIO_OUT_REG to set GPIO0 and
GPIO1
```

This sets both GPIO0 and GPIO1 simultaneously.

4. Addressing Modes in RISC-V

RISC-V supports various addressing modes, and understanding how to use them effectively is crucial when working with memory-mapped registers.

4.1 Immediate Addressing

Immediate addressing involves using a constant value (an immediate) directly in an instruction. This is common when setting or clearing specific bits:

```
li      t1, 0x1              # Immediate value 1 (0000 0001)
```

4.2 Register Addressing

Register addressing involves using a register to hold the address of a memory location. This is useful for accessing hardware registers:

```
lui     t0, 0x60004        # Load base address into t0
```

4.3 Base + Offset Addressing

This mode combines a base address (in a register) with an immediate offset. It's often used to access specific memory locations or registers:

```
addi    t0, t0, 0x020     # Add offset to base address for
GPIO_ENABLE_REG
```

5. Practical Example: Configuring Multiple GPIOs

Let's put everything together in a practical example where we configure multiple GPIOs on the ESP32-C3.

Objective: Configure GPIO0, GPIO2, and GPIO4 as outputs and set them high.

1. Set GPIO0, GPIO2, and GPIO4 as outputs:

```
lui     t0, 0x60004       # Load base address for GPIO registers

addi    t0, t0, 0x020     # Offset to GPIO_ENABLE_REG

li      t1, 0x15          # 0x15 (0001 0101) sets bits 0, 2, and 4

sw      t1, 0(t0)         # Enable GPIO0, GPIO2, and GPIO4 as
outputs
```

2. Set GPIO0, GPIO2, and GPIO4 high:

```
lui     t0, 0x60004       # Load base address for GPIO registers

addi    t0, t0, 0x000     # Offset to GPIO_OUT_REG

li      t1, 0x15          # 0x15 (0001 0101) sets GPIO0, GPIO2, and
GPIO4 high

sw      t1, 0(t0)         # Write to GPIO_OUT_REG to set the pins
high
```

In this example, we efficiently manipulate multiple GPIOs using hexadecimal and bitwise operations, demonstrating how these techniques are applied in real-world scenarios on the ESP32-C3.

6. Conclusion

Hexadecimal and binary are powerful tools in low-level programming, allowing precise control over memory and hardware registers. In this chapter, we've explored how to leverage these representations for bit manipulation and memory manipulation in RISC-V assembly, particularly on the ESP32-C3 microcontroller.

Mastering these concepts is essential for effective embedded programming, as they form the foundation of interacting with and controlling microcontroller hardware. In the next chapter, we will delve into even more advanced topics, including interrupt handling and peripheral control using RISC-V assembly.

By understanding and applying the principles outlined in this chapter, you will be well-equipped to tackle more complex programming challenges in your journey with RISC-V and the ESP32-C3.

CHAPTER 7: REGISTERS

Introduction to RISC-V and ESP32-C3 Registers

In the ESP32-C3, which uses the RISC-V architecture, registers are a critical component for executing instructions and manipulating data. Registers are small, fast storage locations within the CPU, and RISC-V defines 32 general-purpose 32-bit registers (`x0` to `x31`). These registers are used to hold data temporarily during program execution, and understanding their function is key to mastering RISC-V assembly programming for the ESP32-C3.

General-Purpose Registers (`x0` to `x31`)

RISC-V's general-purpose registers are 32 bits wide, allowing for 32-bit data manipulation. They are denoted as `x0` through `x31`, and each register has a specific role and nickname for easier understanding and usage.

Register	Nickname	Description
`x0`	`zero`	Always contains the value 0. Writes to this register are ignored.
`x1`	`ra`	Return address register. Stores the return address for function calls.
`x2`	`sp`	Stack pointer. Points to the top of the stack, used for function calls.
`x3`	`gp`	Global pointer. Points to the global data section of memory.
`x4`	`tp`	Thread pointer. Used to point to thread-local storage.
`x5`	`t0`	Temporary register. Used for temporary storage, not preserved across calls.
`x6`	`t1`	Temporary register. Used for temporary storage, not preserved across calls.
`x7`	`t2`	Temporary register. Used for temporary storage, not preserved across calls.
`x8`	`s0/fp`	Saved register/Frame pointer. Preserved across calls, used in stack frames.
`x9`	`s1`	Saved register. Preserved across calls, used for variables needing persistence.
`x10`	`a0`	Argument register. Holds the first function argument or return value.
`x11`	`a1`	Argument register. Holds the second function argument.
`x12`	`a2`	Argument register. Holds the third function argument.

x13	a3	Argument register. Holds the fourth function argument.
x14	a4	Argument register. Holds the fifth function argument.
x15	a5	Argument register. Holds the sixth function argument.
x16	a6	Argument register. Holds the seventh function argument.
x17	a7	Argument register. Holds the eighth function argument.
x18	s2	Saved register. Preserved across calls, used for variables needing persistence.
x19	s3	Saved register. Preserved across calls, used for variables needing persistence.
x20	s4	Saved register. Preserved across calls, used for variables needing persistence.
x21	s5	Saved register. Preserved across calls, used for variables needing persistence.
x22	s6	Saved register. Preserved across calls, used for variables needing persistence.
x23	s7	Saved register. Preserved across calls, used for variables needing persistence.
x24	s8	Saved register. Preserved across calls, used for variables needing persistence.
x25	s9	Saved register. Preserved across calls, used for variables needing persistence.
x26	s10	Saved register. Preserved across calls, used for variables needing persistence.
x27	s11	Saved register. Preserved across calls, used for variables needing persistence.
x28	t3	Temporary register. Used for temporary storage, not preserved across calls.
x29	t4	Temporary register. Used for temporary storage, not preserved across calls.
x30	t5	Temporary register. Used for temporary storage, not preserved across calls.
x31	t6	Temporary register. Used for temporary storage, not preserved across calls.

Each of these registers plays a specific role in assembly language programs, and it’s important to use them properly to ensure your code is efficient and bug-free. For example, registers `x5` through `x7`, and `x28` through `x31` are considered temporary registers, and their contents may be overwritten during function calls, so they should only be used for short-term storage. On the other hand, registers `x8`, `x9`, and `x18` through `x27` are saved registers, meaning they must be preserved across function calls. This preservation is typically handled by the calling convention used in your programs.

Conclusion

Mastering the usage of 32-bit general-purpose registers is fundamental to writing efficient RISC-V assembly code for the ESP32-C3. Whether you're manipulating data, controlling program flow, or managing function calls, the correct use of registers ensures smooth and optimized execution. In future chapters, we will explore more advanced topics such as interrupt handling and memory access using these registers.

CHAPTER 8: ADD

In this chapter, we'll explore how to perform addition using assembly language in RISC-V for the ESP32-C3 microcontroller. We'll start by understanding the basics of RISC-V assembly instructions for arithmetic operations and then demonstrate how to write an assembly function that adds two numbers. Finally, we'll show how to call this assembly function from a C program.

Introduction

The ESP32-C3 is a low-power microcontroller that utilizes the RISC-V architecture. Writing assembly code for the ESP32-C3 can provide performance benefits and a deeper understanding of how the hardware operates. In this chapter, we'll focus on creating an assembly function that adds two integers and returns the result.

RISC-V Assembly Basics

Before diving into the code, let's review some basic concepts:

- **Registers**: RISC-V has 32 general-purpose registers (x0 to x31), each 32 bits wide in RV32I. Registers x5 to x7 are designated as temporary registers (t0 to t2), which we can use for intermediate calculations.
- **Instructions**: RISC-V assembly instructions are simple and consistent, making them easier to learn and use.

Writing the Assembly Function

Let's write an assembly function named add_numbers that takes two integers as input and returns their sum.

```
.global add

add:
    add     a0, a0, a1   # add the values in a0 and a1, store result
in a0
    ret                  # return to caller
```

In our next chapter we will debug this so you can see exactly what is going on inside the registers and memory.

CHAPTER 9: DEBUGGING ADD

In this chapter, we'll explore how to debug addition using assembly language in RISC-V for the ESP32-C3 microcontroller.

Step 1: Set Breakpoint

Let's open our **main.S** file within VS Code and PlatformIO and put a breakpoint on the add instruction. You will click in the left-hand side of the screen and a red dot will appear.

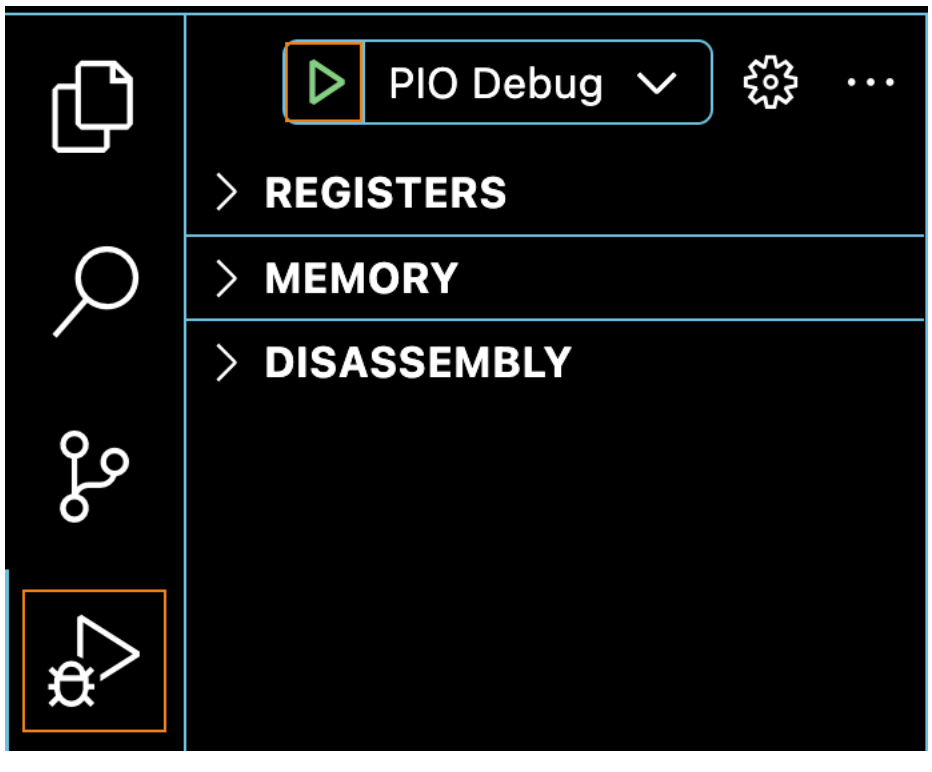
```
.global add

add:
*   add     a0, a0, a1   # add the values in a0 and a1, store result
in a0
    ret                  # return to caller
```

Step 2: Compile and Flash

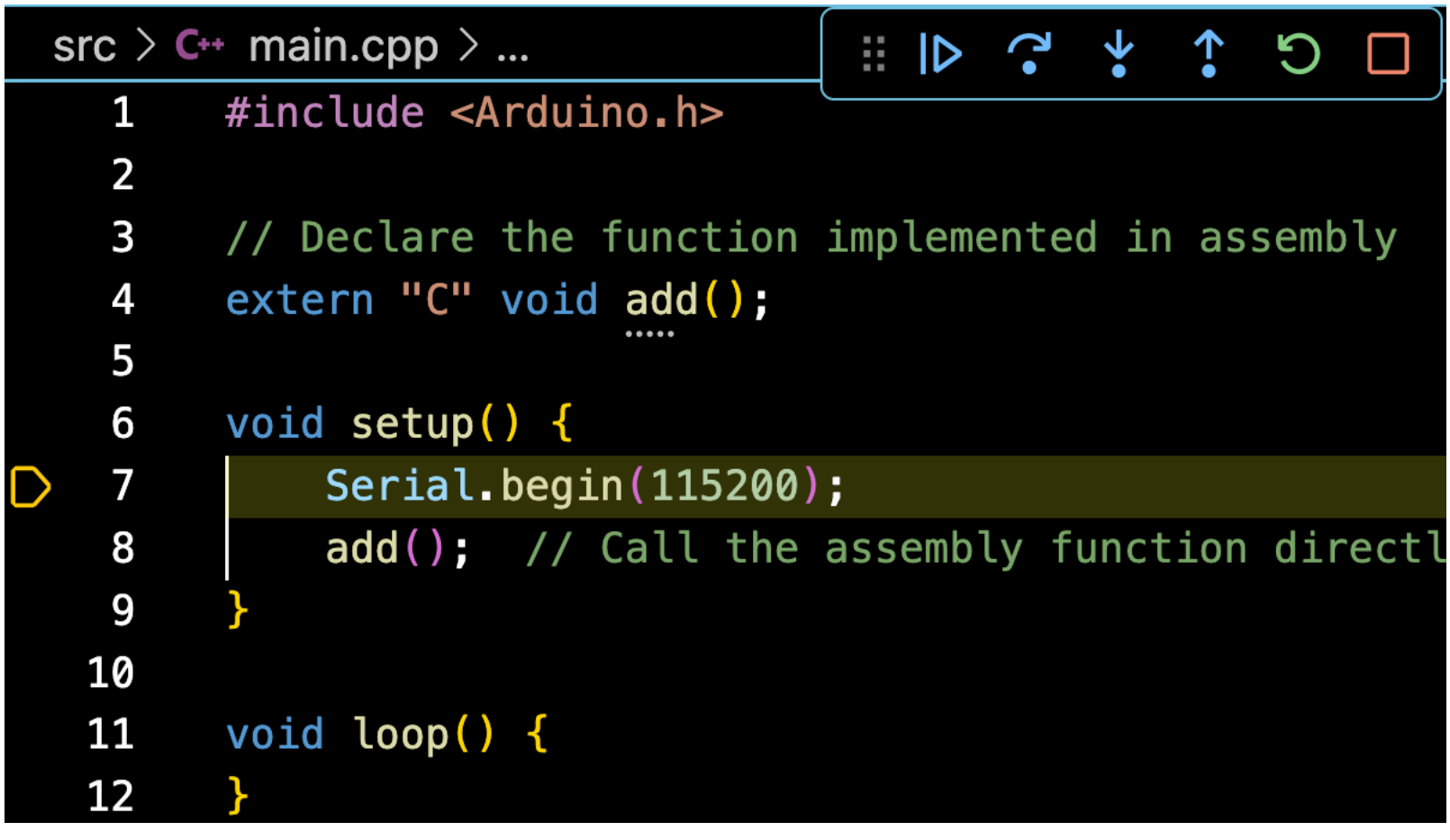
At the bottom of the screen you will see a checkmark and right arrow, click the checkmark to compile and the right arrow to flash.

Step 3: Debug

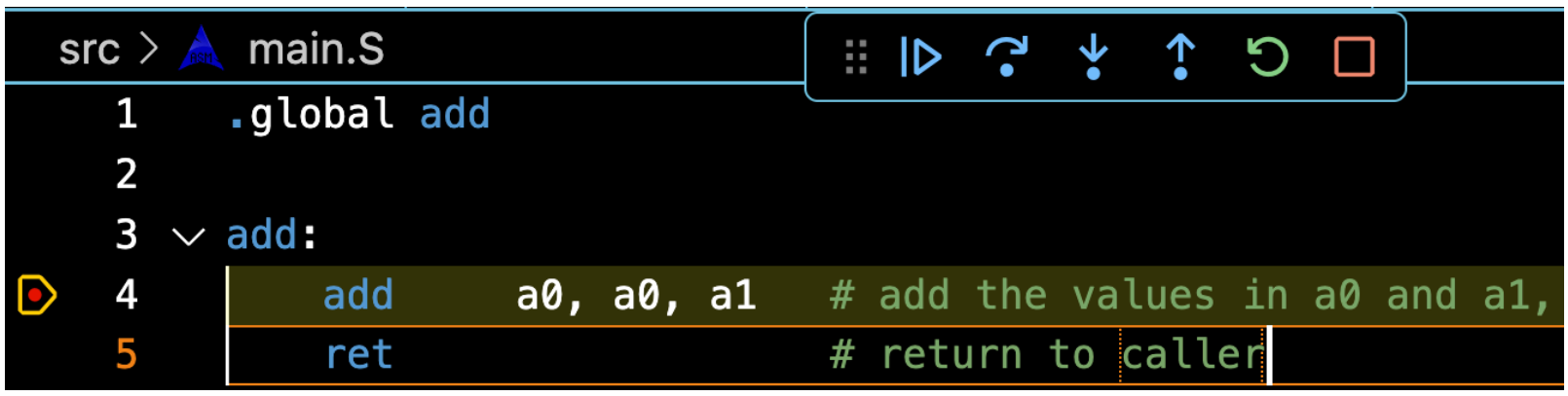
Open up the debug panel and begin the debug process by pressing the green arrow.



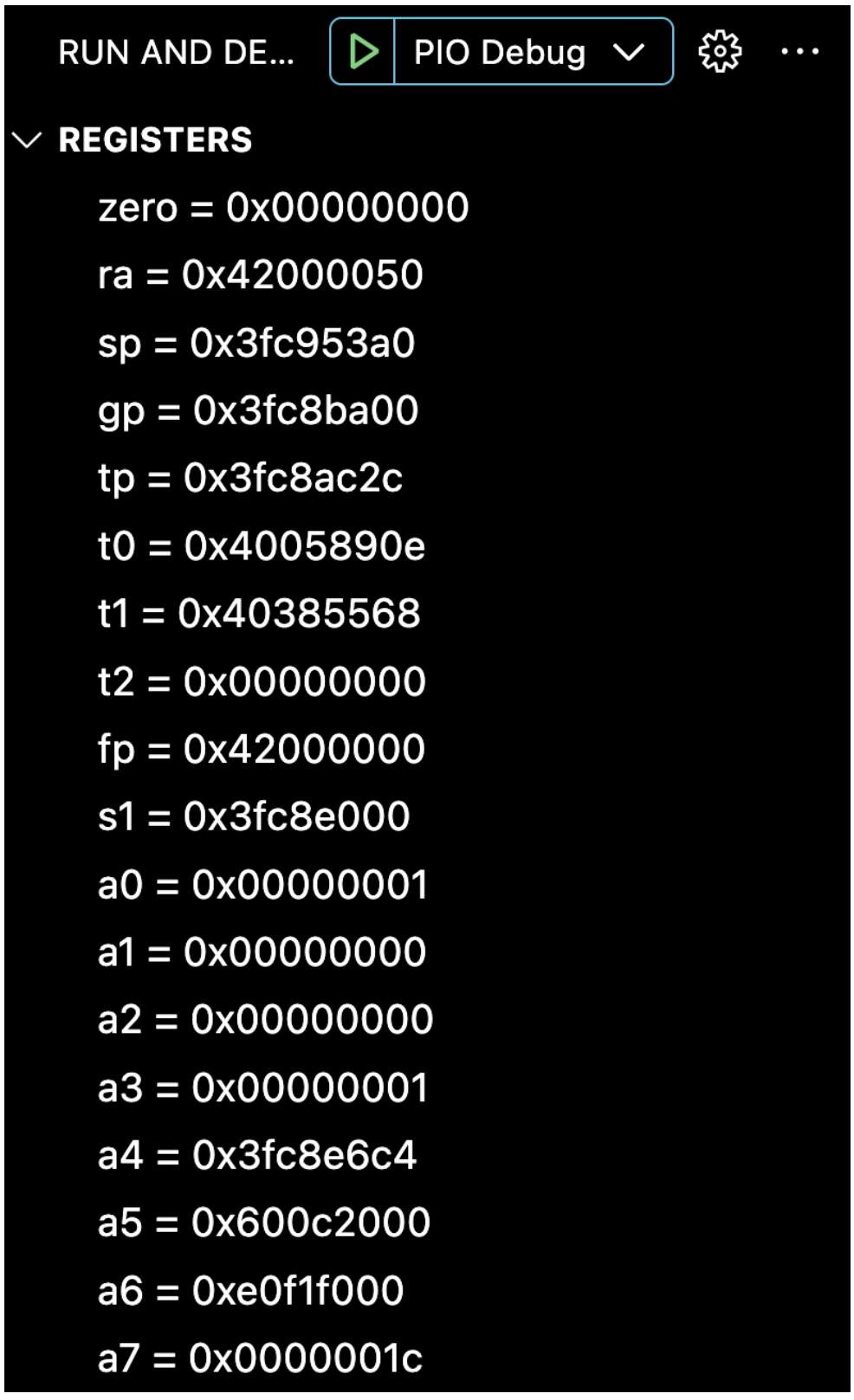
Our entry point is is the setup area. We will press F5 or press the play button to take us to our breakpoint.



At this point see see the assembler breakpoint.



Let’s examine the processor registers.



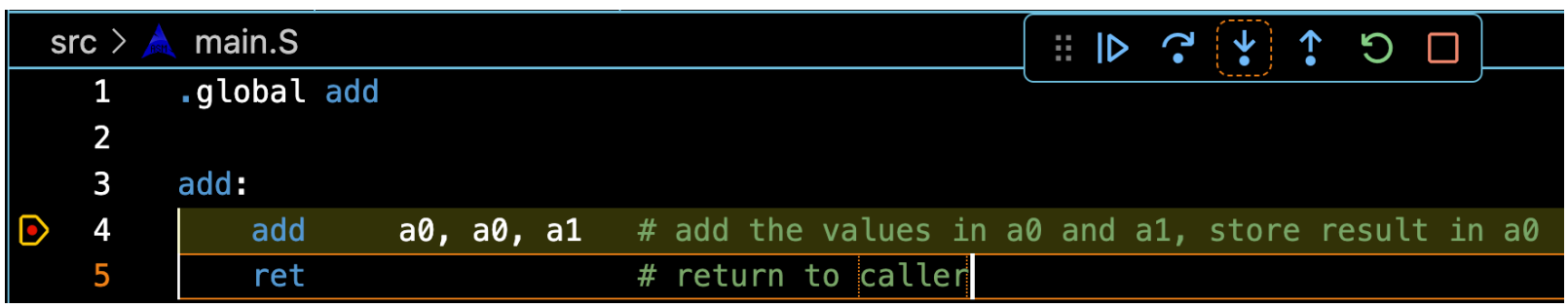
Now I want you to look at the code and think about what it is going to do.

```
add     a0, a0, a1
```

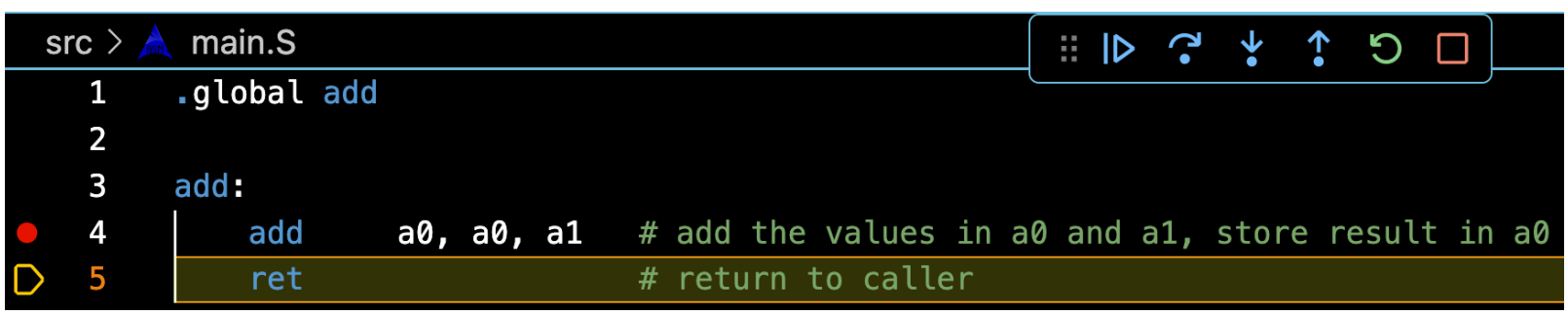
I deliberately created this non-sensical code to show you that we are putting WHATEVER is in *a1* into *a0* without actual moving in a deterministic value into *a1*.

We see, in this case, *a1* has 0 therefore there will be NO CHANGE when we step forward in assembler.

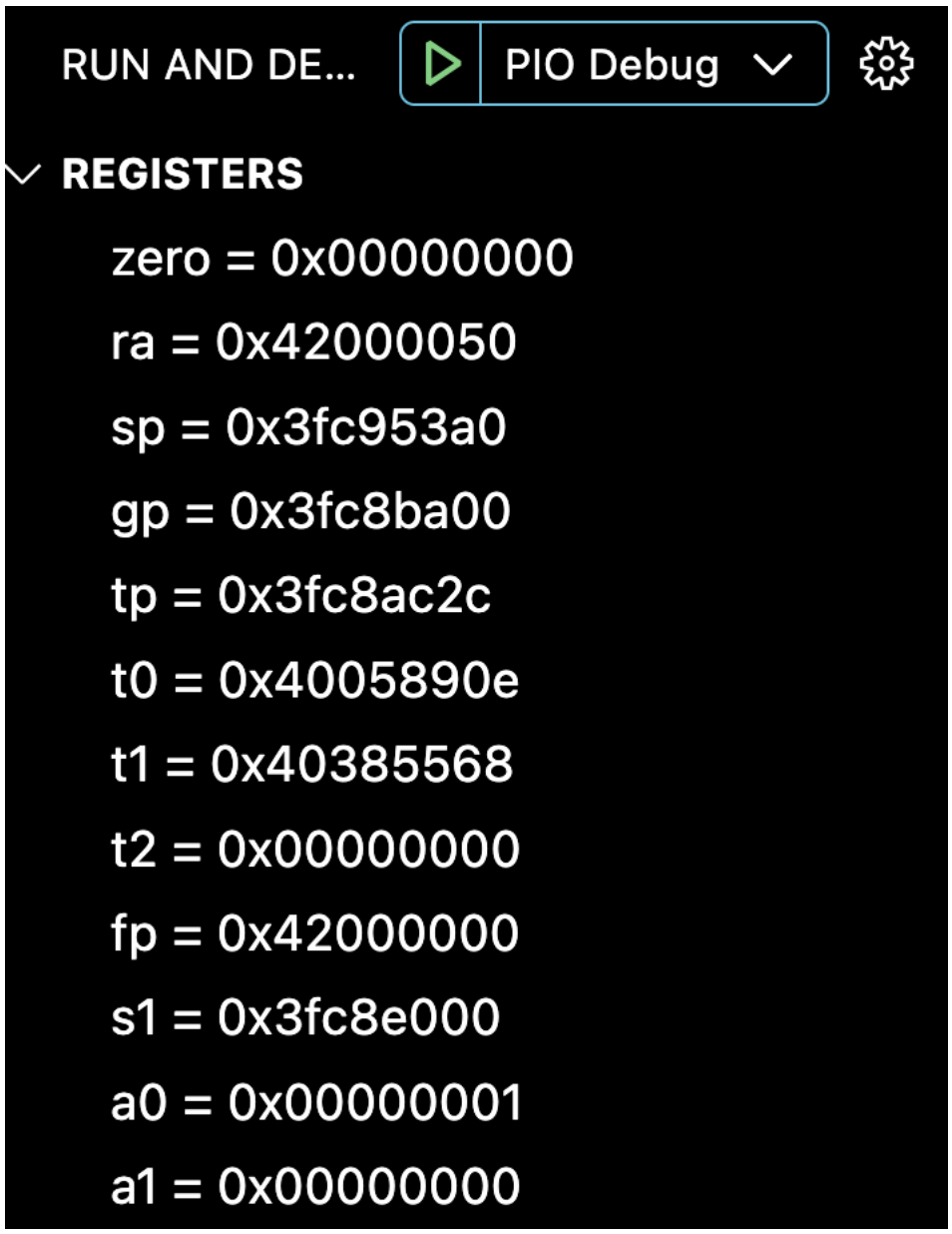
Lets step-into and re-observe the registers.



Now we are about the execute the ret statement however.



Let’s re-examine the registers and verify no change has occurred.



In our next lesson, we will hack the values in real-time within the debug console!

```
set $a1 = 42
```