Q.1: 1 Install & Sanity-Check the Toolchain

Question to ask AI tools

“I have downloaded riscv-toolchain-rv32imac-x86\_64-ubuntu.tar.gz. How exactly do I unpack it, add it to PATH, and confirm the gcc, objdump, and gdb binaries work?”

What AI tool should return / what you should do

• Step-by-step tar -xzf ... command.

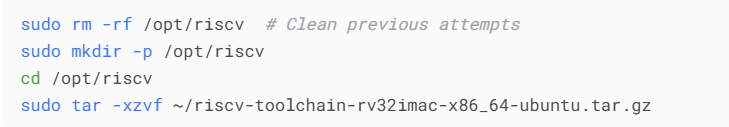
• export PATH=$HOME/riscv/bin:$PATH lines for ~/.bashrc.

• Verification commands (riscv32-unknown-elf-gcc --version, etc.).

ANS: 1. UNPACKING THE TOOLCHAIN:

COMMANDS:

* **Step 1: Unpack the Toolchain**



Verifying the extraction:

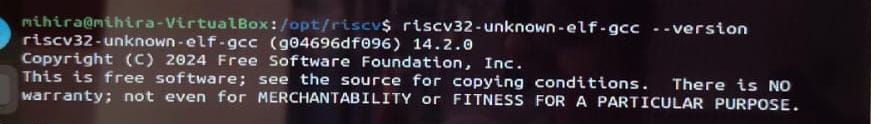


* Step 2: Add to Path:

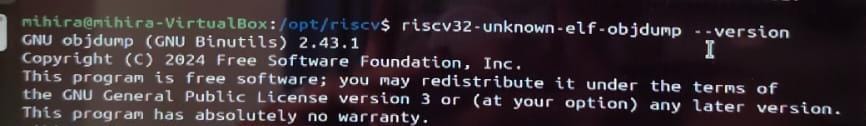


* Step 3: Sanity-Check All Binaries:

1. Command:

Expected Output: 

1. Command:

Expected Output: 

Q.2: Create a Hello C Program

Ans:

* **STEP-1: Write a minimal C program**

1. **Create a file named nano hello.c in the terminal and write the following program.**



This opens a simple text editor in the terminal.

Paste this code:

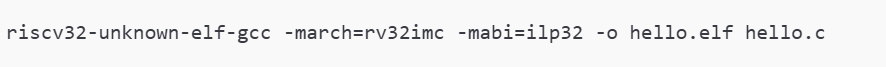


**Save and Exit:**

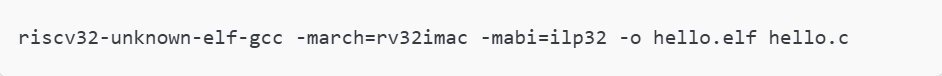
Press Ctrl + O to save, Press Enter to confirm filename, Press Ctrl + X to exit

* **Step 2: Compile using the RISC-V cross-compiler:**

Run this command in the terminal:



If you’re getting error, then try this instead:

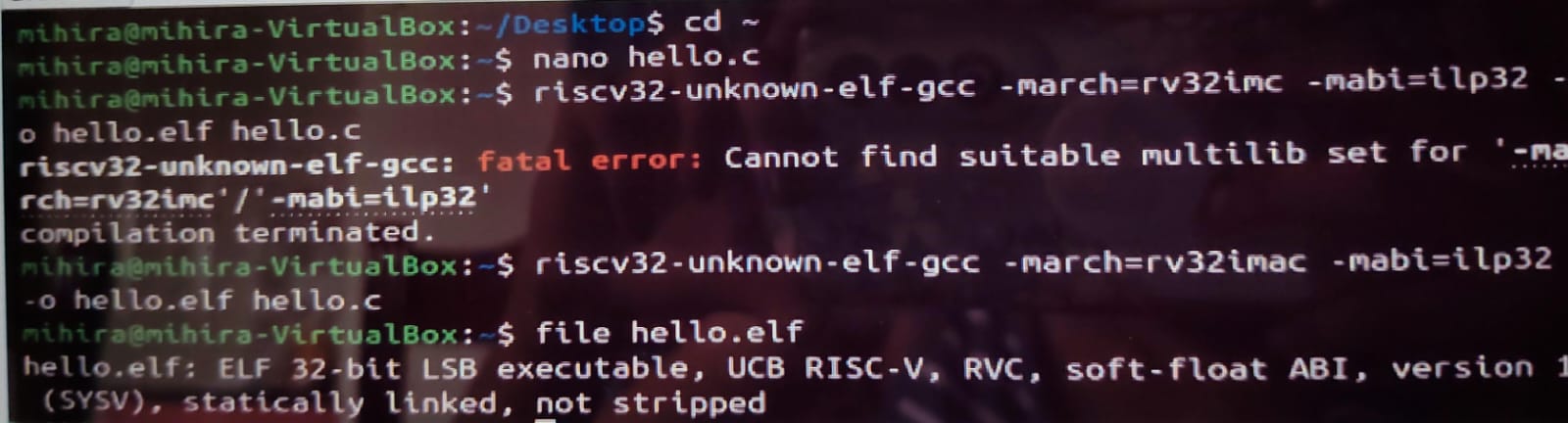


* **Step 3: Verify the output ELF file:**

Then run this instruction in the terminal:



Expected Output:

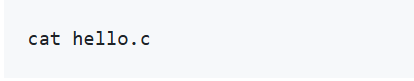
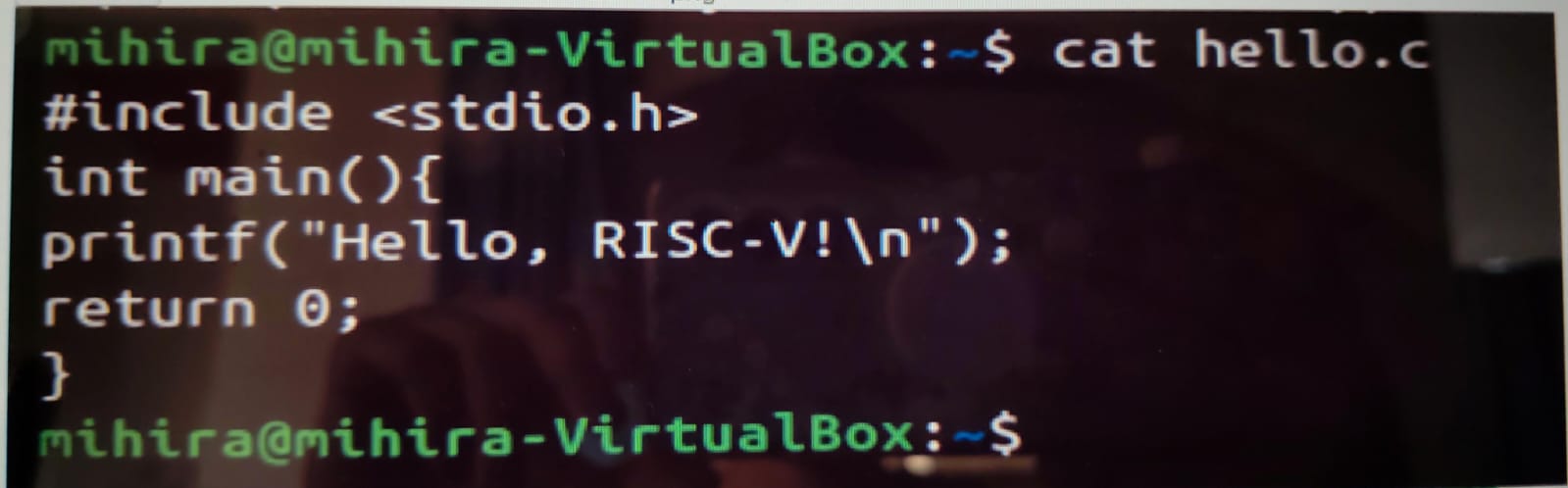


**Q3: From C to Assembly**

**Ans: GOAL:**

Understand how a simple C program like main() is translated into RISC-V assembly code, and learn what the function prologue and epilogue do.

* Step-1 : **Ensure your C file still exists:**



* **Step -2: Generate the assembly code:**

Use the RISC-V cross-compiler to generate the .s file (assembly):



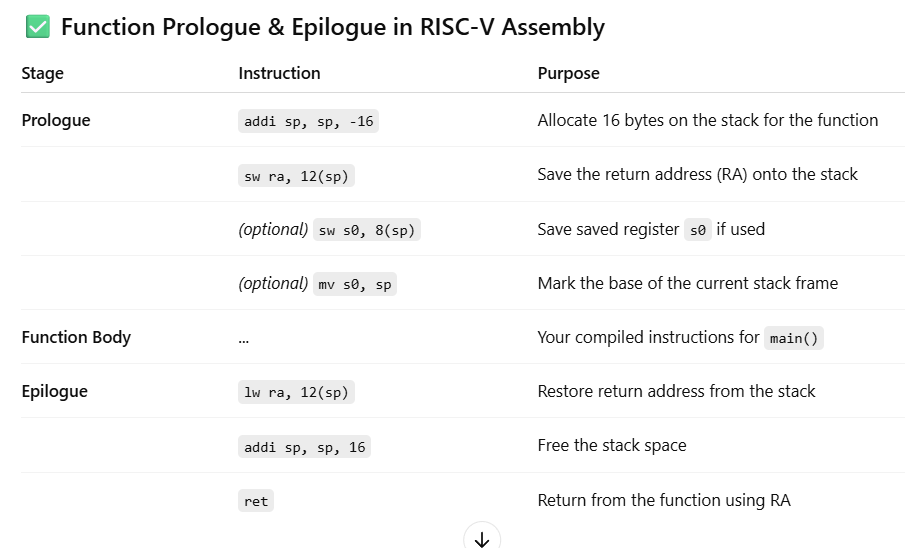
This will create: hello.s

* **Step -3: View the Assembly File:**

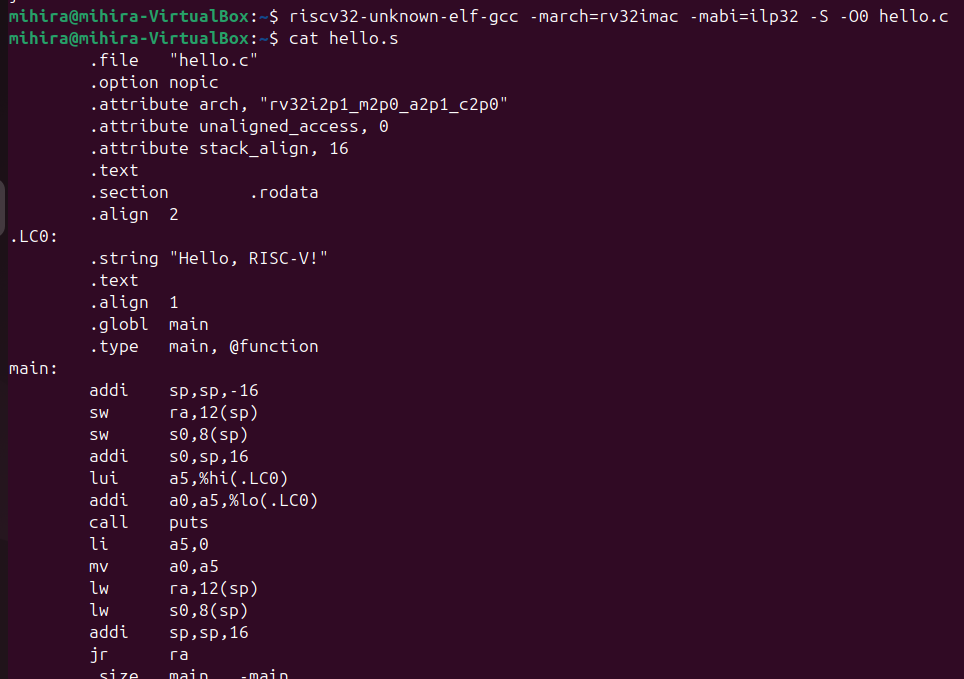


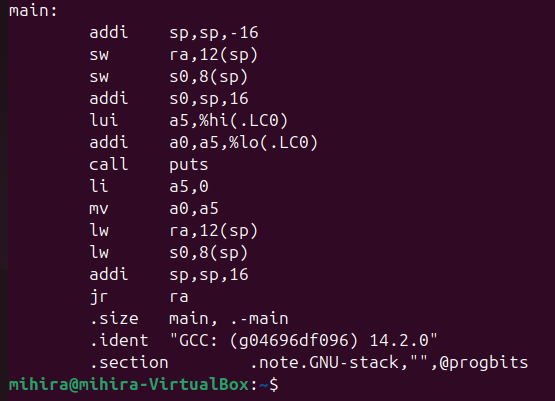
Look at the assembly instructions inside the main: function.

* **Step 4: Identify and Understand the Prologue and Epilogue:**

****

Expected Output:

****

****

**Q4: Hex Dump & Disassembly:**

**Ans: Goal:**

Disassemble the compiled hello.elf to see the actual RISC-V instructions. Convert it to Intel HEX format (raw memory content). Learn what each column in the disassembly means

* **Step – 1: 1. Disassemble the ELF File:**

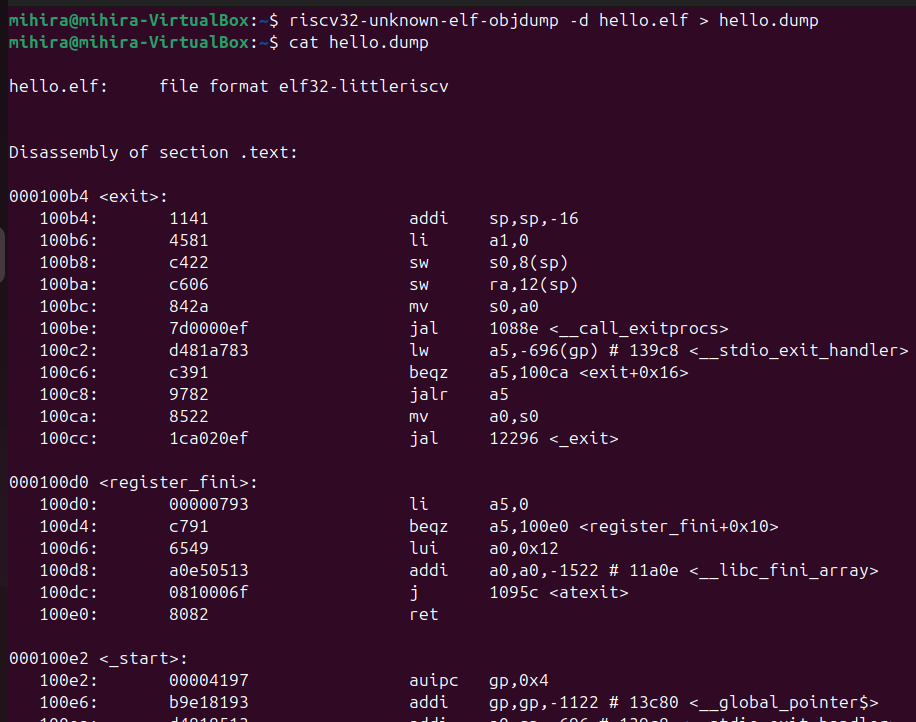
**Run:**

This command uses objdump to disassemble the ELF file and dumps the output into a readable text file called hello.dump

**You can view it:**

****

**Expected output (Partial) :**

****

* **Step – 2: Convert ELF to HEX format (raw memory):**

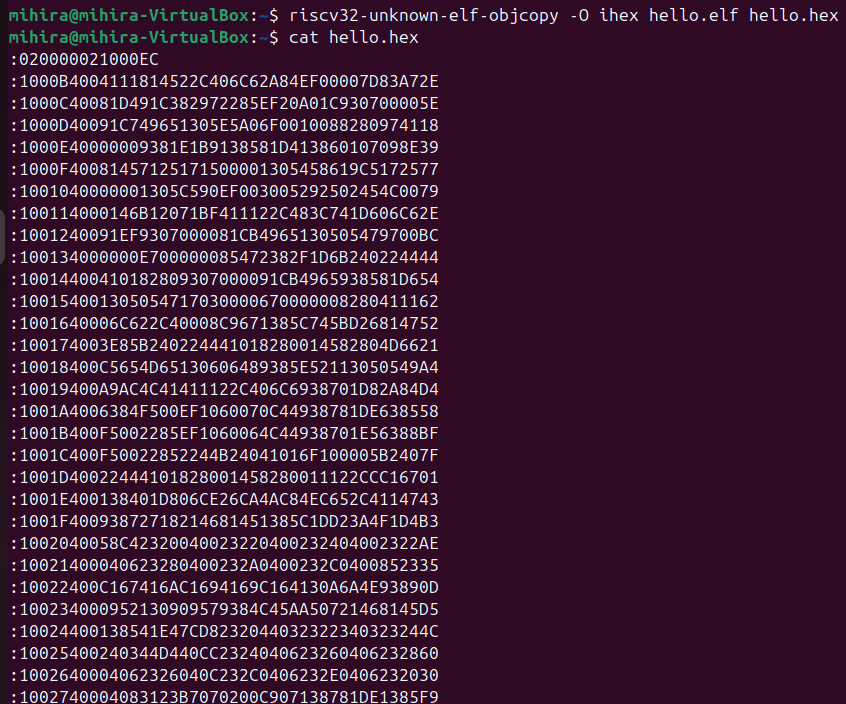
**Run:**

****

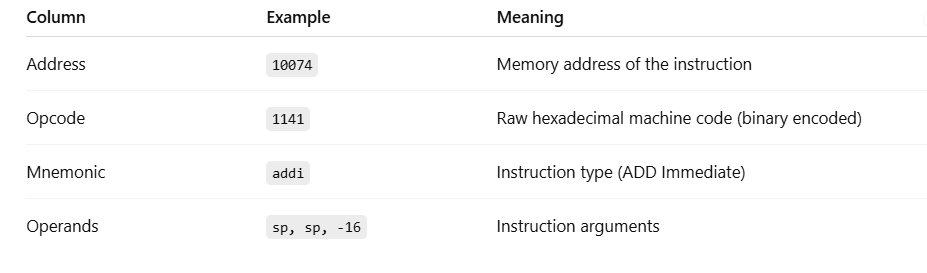
This creates hello.hex, a file in Intel HEX format (used for loading binaries into hardware or emulators).

**You can view it with:**

****

**Expected Output (Partial): **

**Understanding the Disassembly Output:**

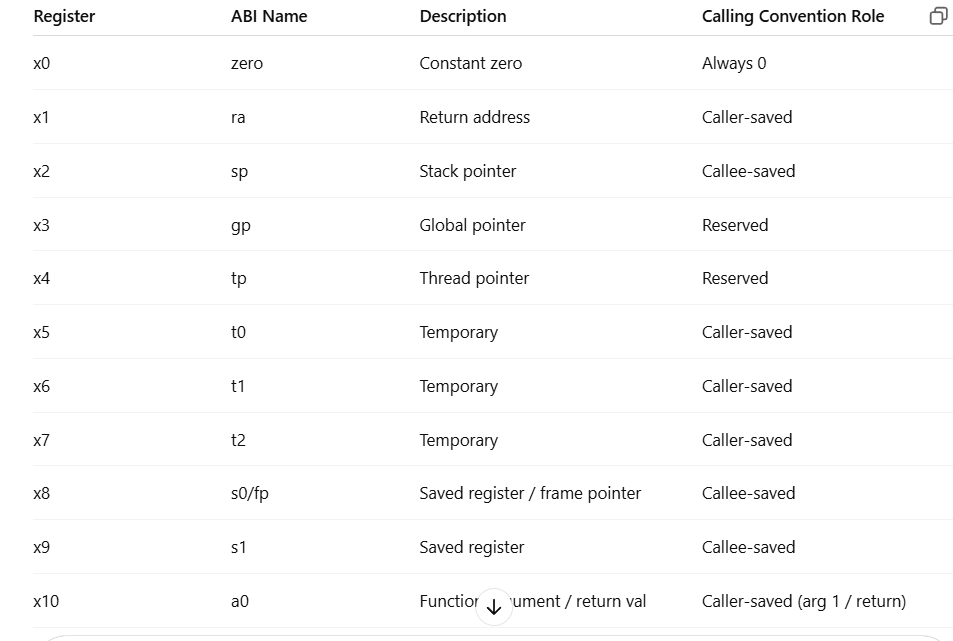
****

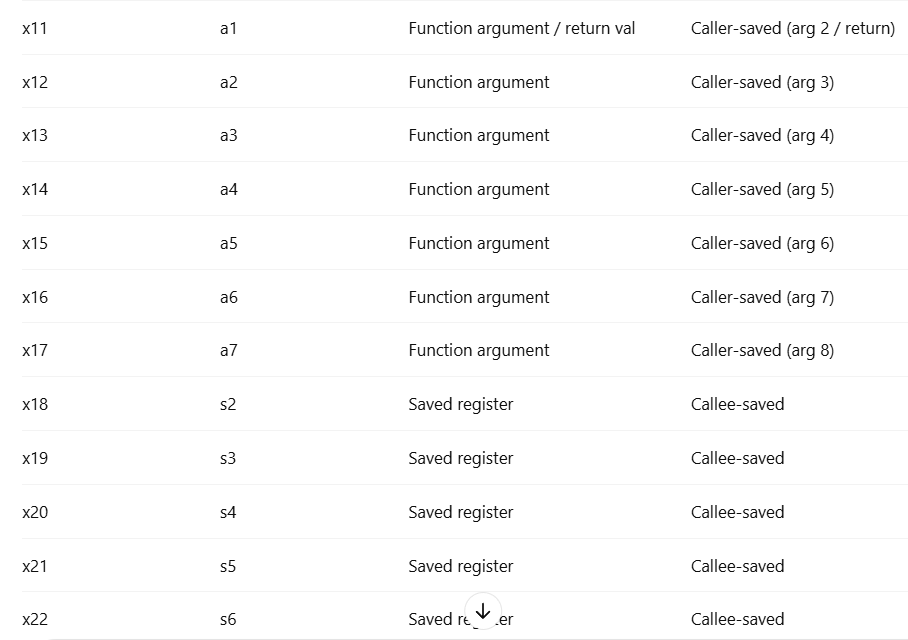
**Q5: ABI & Register Cheat-Sheet:**

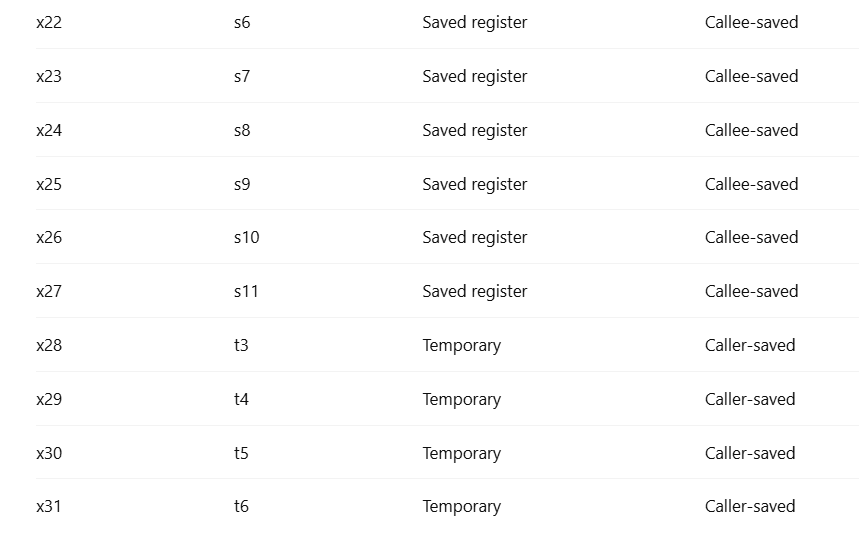
**Ans: GOAL:**

Learn the names and roles of all 32 general-purpose registers in RISC-V (RV32). Understand which registers are used for function arguments, return values, and preservation across calls.

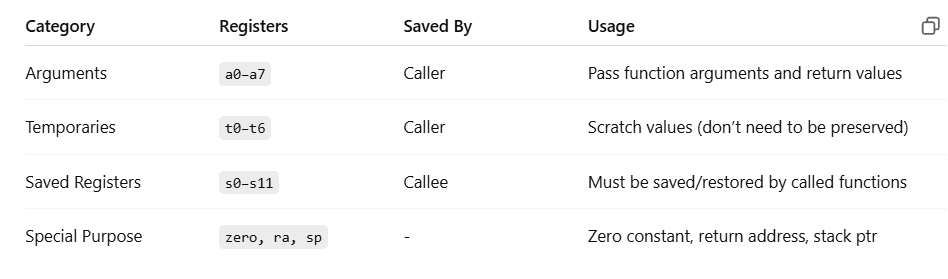
* **Full Register Mapping (x0–x31):**







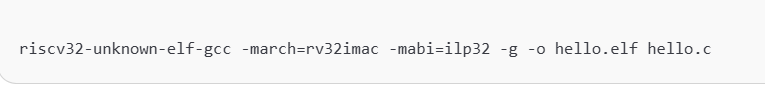
* Calling Convention Summary:



Q.6: Stepping with GDB:

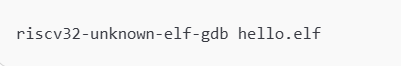
Ans: Goal:

* Debug your RISC-V binary using riscv32-unknown-elf-gdb
* Set breakpoints, step through code, and inspect registers
* Understand what's happening in low-level machine execution
* **Step – 1: Compile your program with debug symbols:**

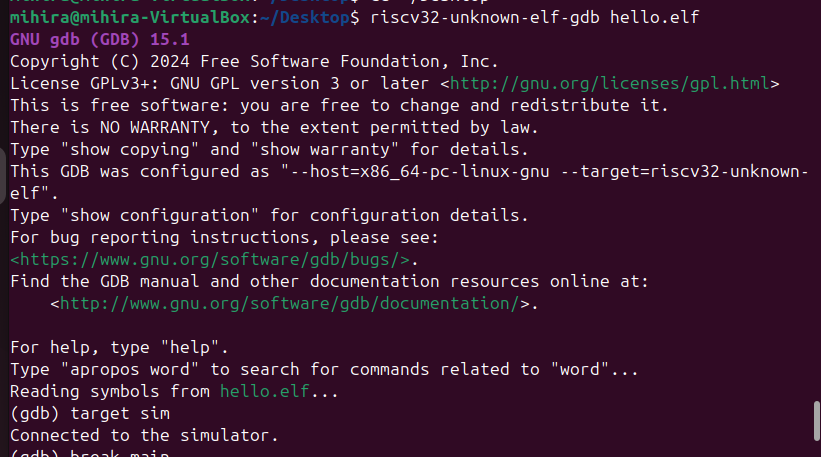


-g: Includes debug info for GDB and -o hello.elf: Output file

* **Step – 2: Start GDB on your ELF:**



Output:



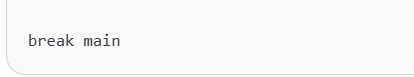
* **Step – 3: Inside GDB: Set up the simulation:**

At the gdb prompt, enter:



This connects GDB to the built-in RISC-V simulator.

* **Step – 4: Set a breakpoint at main:**

****

* **Step – 5: Run the program:**

****

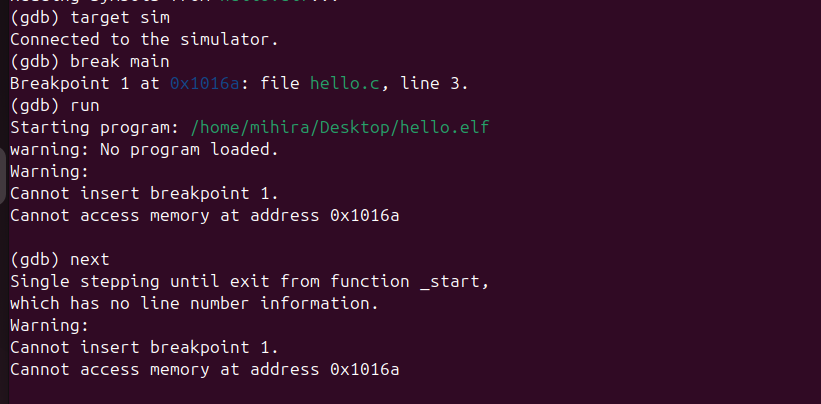
You’ll hit the breakpoint at the start of main().

* **Step – 6: Step through the program:**

****

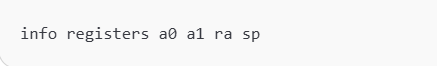
to step over the next C statement.

Output:

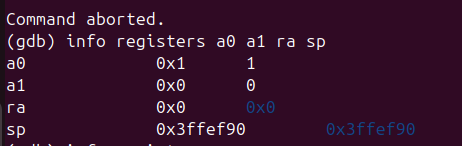


* **Step – 7: Inspect registers:**

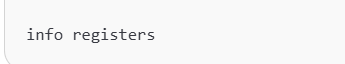
**Check specific registers:**

****

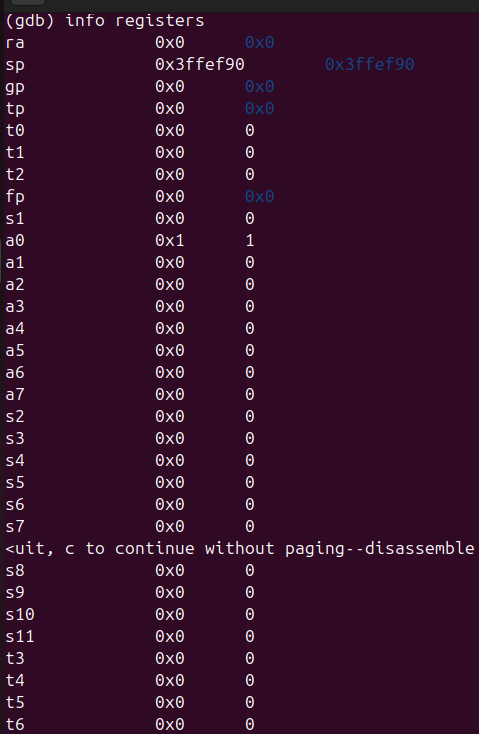
**Output:**

****

**Or view all general-purpose registers:**

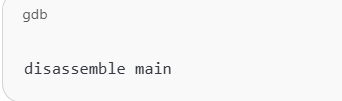
****

**Ouput:**

****

* **Step – 8: Disassemble your function:**

**To see the assembly around main():**

****

**Q.7: Running Under an Emulator:**

**Ans: GOAL:**

Run your hello.elf file on an emulator (Spike or QEMU) and view the UART console output — i.e., your printf() from C.

* STEP -1: Run with QEMU:

**Prerequisites:**

1. QEMU installed (qemu-system-riscv32)



1. A minimal bare-metal hello.elf that doesn't rely on Linux syscalls (only direct memory-mapped UART)