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.).

Response:
Here's a **step-by-step guide** for unpacking the RISC-V toolchain, adding it to your `PATH`, and verifying that it works:

✅ Step 1: Unpack the Toolchain Archive

Open your terminal and run the following:

```
tar -xzf riscv-toolchain-rv32imac-x86_64-ubuntu.tar.gz
```

This will extract the contents into a directory, usually named something like `riscv` or similar.

> 🔎 If you're not sure where it extracted, run `ls` and look for a new folder — commonly `riscv`.

✅ Step 2: Add to Your `PATH`

Assuming the extracted folder is `riscv` in your home directory (`$HOME/riscv`), add this to your `PATH`.

Edit your `~/.bashrc`:

```
nano ~/.bashrc
```

Add the following line at the end:

```
export PATH="$HOME/riscv/bin:$PATH"
```

Save and close (`Ctrl+O`, `Enter`, then `Ctrl+X`), and reload the shell:

```
source ~/.bashrc
```

✅ Step 3: Verify the Toolchain Works

Run these commands to confirm the toolchain is available:

```
riscv32-unknown-elf-gcc --version
riscv32-unknown-elf-objdump --version
riscv32-unknown-elf-gdb --version
```

You should see version info for each tool.

🧠 Notes:

- If the extracted folder is in a different location, adjust the `PATH` accordingly.
- If you're using `zsh`, update `~/.zshrc` instead of `~/.bashrc`.

Problems faced:

The error you're seeing:

```
riscv32-unknown-elf-gdb: error while loading shared libraries: libpython3.10.so.1.0: cannot open shared object file: No such file or directory
```

means that the version of `riscv32-unknown-elf-gdb` you downloaded was built against **Python 3.10**, but your system does **not have** the required shared library (`libpython3.10.so.1.0`) installed.

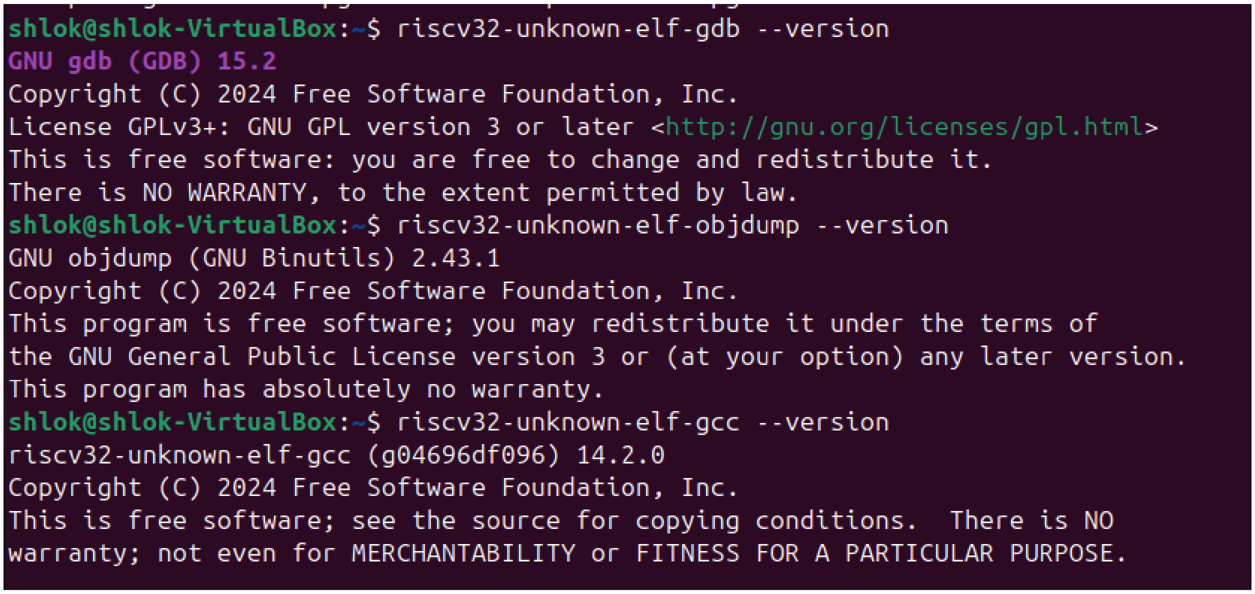
✅ Option 1: Install Python 3.10 and Its Development Files

If you're okay installing Python 3.10, do this:

```
sudo apt update

sudo apt install python3.10 python3.10-dev
```

Then try running `riscv32-unknown-elf-gdb` again.

OUTPUT:



```
shlok@shlok-VirtualBox:~$ riscv32-unknown-elf-gdb --version
GNU gdb (GDB) 15.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
shlok@shlok-VirtualBox:~$ riscv32-unknown-elf-objdump --version
GNU objdump (GNU Binutils) 2.43.1
Copyright (C) 2024 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or (at your option) any later version.
This program has absolutely no warranty.
shlok@shlok-VirtualBox:~$ riscv32-unknown-elf-gcc --version
riscv32-unknown-elf-gcc (g04696df096) 14.2.0
Copyright (C) 2024 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

---

2 Compile “Hello, RISC-V”

Question

“Show me a minimal C ‘hello world’ that cross-compiles for RV32IMC and the exact gcc flags to produce an ELF.”

Expected Method

- Copy the one-line printf("Hello…") program.
- Run riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -o hello.elf hello.c.
- Use file hello.elf to confirm it’s 32-bit RISC-V.

Response:

Here's how to write and cross-compile a minimal **"Hello, RISC-V"** C program for the **RV32IMC** target:

✅ Step 1: Minimal C Program

Create a file called `hello.c` with:

```
#include <stdio.h>


int main() {
    printf("Hello, RISC-V!\n");
    return 0;
}
```

✅ Step 2: Cross-Compile for RV32IMC

Use the RISC-V GCC toolchain:

```
riscv32-unknown-elf-gcc -march=rv32imc -mabi=ilp32 -o hello.elf hello.c
```

- `-march=rv32imc`: Target the RV32IMC architecture.
- `-mabi=ilp32`: Use the integer-only ABI for 32-bit RISC-V.

> 🔍 This creates `hello.elf`, a RISC-V executable (not runnable on x86).

✅ Step 3: Confirm ELF Target

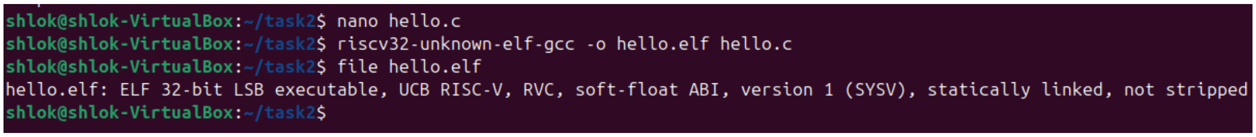
Check the file type:

```
file hello.elf
```

Expected output (or similar):

```
hello.elf: ELF 32-bit LSB executable, UCB RISC-V, ...
```

OUTPUT:



```
shlok@shlok-VirtualBox:~/task2$ nano hello.c
shlok@shlok-VirtualBox:~/task2$ riscv32-unknown-elf-gcc -o hello.elf hello.c
shlok@shlok-VirtualBox:~/task2$ file hello.elf
hello.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked, not stripped
shlok@shlok-VirtualBox:~/task2$
```

---

3 From C to Assembly

Question

“How do I generate the .s file and explain the prologue/epilogue of the main function?”

Method

• riscv32-unknown-elf-gcc -S -O0 hello.c generates hello.s.

• ChatGPT explains why you see addi sp,sp,-16, sw ra,12(sp)…

RESPONSE:

Here's how you go from a C program to RISC-V assembly, and an explanation of the **function prologue and epilogue**:

✅ Step 1: Generate Assembly from C

Given your `hello.c`:

```
#include <stdio.h>


int main() {
    printf("Hello, RISC-V!\n");
    return 0;
}
```

Compile to assembly with:

```
riscv32-unknown-elf-gcc -S -O0 hello.c
```

- `-S`: Output assembly (`.s`) instead of object code.
- `-O0`: Disable optimizations to keep the code readable.

This creates a file `hello.s`.

✅ Step 2: Look at the Function Prologue & Epilogue

Inside `hello.s`, you'll find something like this in the `main` function:

```
main:
    addi   sp, sp, -16      # Allocate 16 bytes on the stack
    sw     ra, 12(sp)       # Save return address
    sw     s0, 8(sp)        # Save frame pointer (callee-saved)
    addi   s0, sp, 16       # Set up new frame pointer
    ...
    lw     ra, 12(sp)       # Restore return address
    lw     s0, 8(sp)        # Restore frame pointer
    addi    sp, sp, 16        # Deallocate stack frame
    ret                     # Return to caller
```

🧠 Explanation

▶️ Prologue: Set up the stack frame

```
addi    sp, sp, -16
```

- Moves the stack pointer down by 16 bytes to allocate space for saved registers.

```
sw      ra, 12(sp)
sw      s0, 8(sp)
```

- Saves `ra` (return address) and `s0` (frame pointer) to the stack.
- RISC-V calling convention requires callee to save `s0`.

```
addi    s0, sp, 16
```

- Initializes the new frame pointer (`s0` = old `sp`).

◀️ Epilogue: Tear down the stack frame

```
lw      ra, 12(sp)
lw      s0, 8(sp)
```

- Restores saved `ra` and `s0`.

```
addi    sp, sp, 16
```

- Frees the 16-byte stack frame.

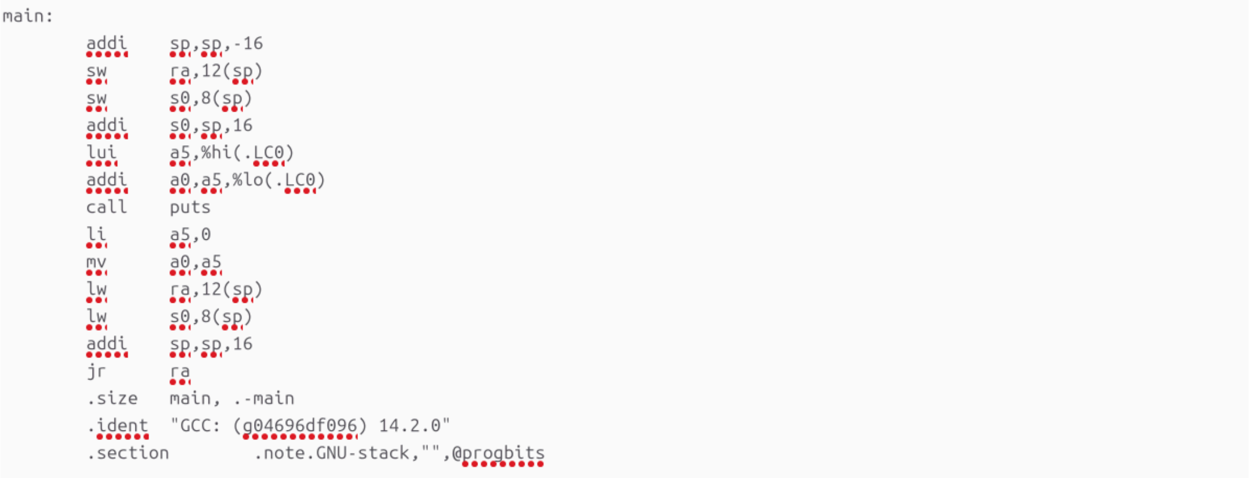
```
ret
```

- Returns to the caller using the restored `ra`.

🧩 Summary

Instruction	Purpose
`addi sp, sp, -16`	Allocate stack space
`sw ra, 12(sp)`	Save return address
`sw s0, 8(sp)`	Save old frame pointer
`addi s0, sp, 16`	Set up new frame pointer
`lw ...` + `ret`	Clean up and return

OUTPUT:



```
main:
        addi    sp,sp,-16
        sw      ra,12(sp)
        sw      s0,8(sp)
        addi    s0,sp,16
        lui     a5,%hi(.LC0)
        addi    a0,a5,%lo(.LC0)
        call    puts
        li      a5,0
        mv      a0,a5
        lw      ra,12(sp)
        lw      s0,8(sp)
        addi    sp,sp,16
        jr      ra
        .size   main, .-main
        .ident  "GCC: (g04696df096) 14.2.0"
        .section        .note.GNU-stack,"",@progbits
```

4 Hex Dump & Disassembly

Question

“Show me how to turn my ELF into a raw hex and to disassemble it with objdump. What do each columns mean?”

Method

- riscv32-unknown-elf-objdump -d hello.elf > hello.dump.
- riscv32-unknown-elf-objcopy -O ihex hello.elf hello.hex.
- Walk through one instruction field (address, opcode, mnemonic, operands).

RESPONE:

Here's how to turn your `hello.elf` binary into both a **disassembly listing** and a **hex dump**, and how to **understand each part** of the output.

✅ Step 1: Disassemble the ELF

To disassemble the RISC-V ELF and save the output:

```
riscv32-unknown-elf-objdump -d hello.elf > hello.dump
```

This produces a human-readable disassembly in `hello.dump`.

✅ Step 2: Generate a Hex Dump

To convert your ELF to **Intel HEX format** (used by programmers or flashers):

```
riscv32-unknown-elf-objcopy -O ihex hello.elf hello.hex
```

You can also use `-O binary` for raw binary, or `-O srec` for Motorola S-records.

✅ Step 3: Understand the Disassembly Output

Example snippet from `hello.dump`:

```
00010074 <main>:
  10074: 1141          addi sp,sp,-16
  10076: c606          sw ra,12(sp)
  10078: c422          sw s0,8(sp)
```

```
   1007a: 842a                addi s0,sp,16
```

Let’s break one line down:

```
10074: 1141    addi sp,sp,-16
```

🔍 Column Breakdown:

Part	Meaning
`10074:`	Memory **address** of instruction
`1141`	**Raw opcode** (in hex) for the instruction
`addi sp,sp,-16`	**Mnemonic and operands**

- `addi` is the instruction name.
- `sp, sp, -16` are the operands: destination and two sources (dest = src + imm).

This shows that at address `0x10074`, an `addi` instruction sets up a stack frame.

🧠 Bonus Tips

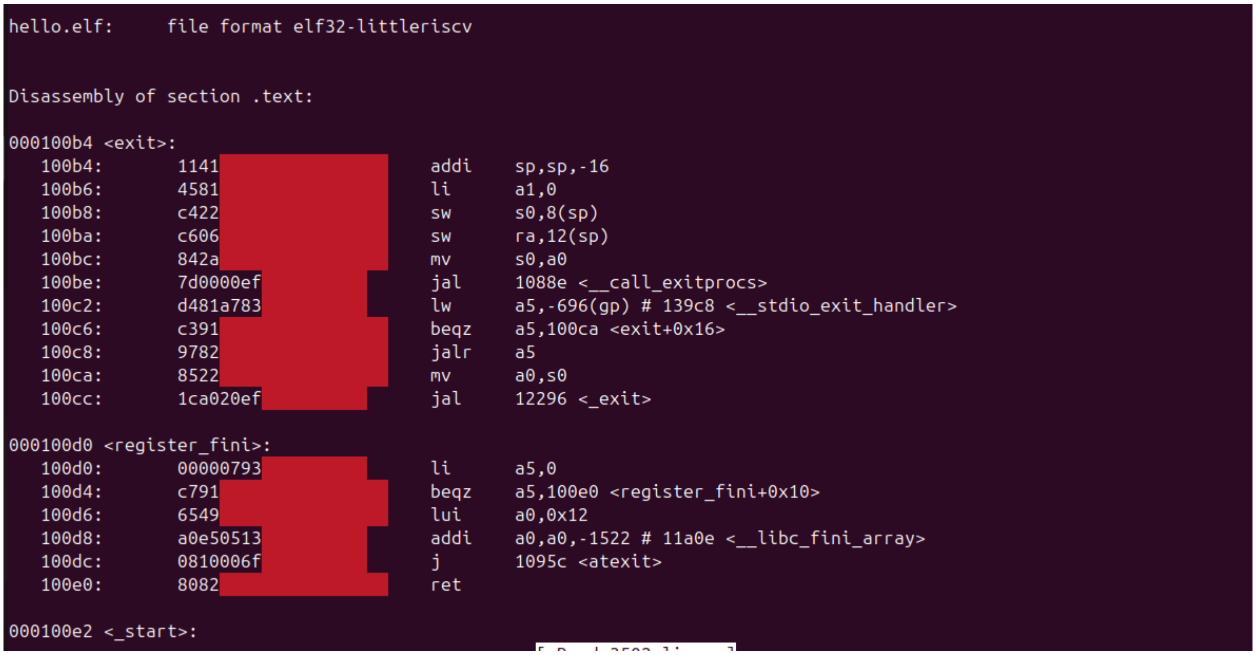
Use `-M no-aliases` to see the *true* base instructions (without pseudoinstructions):

```
 riscv32-unknown-elf-objdump -d -M no-aliases hello.elf
```

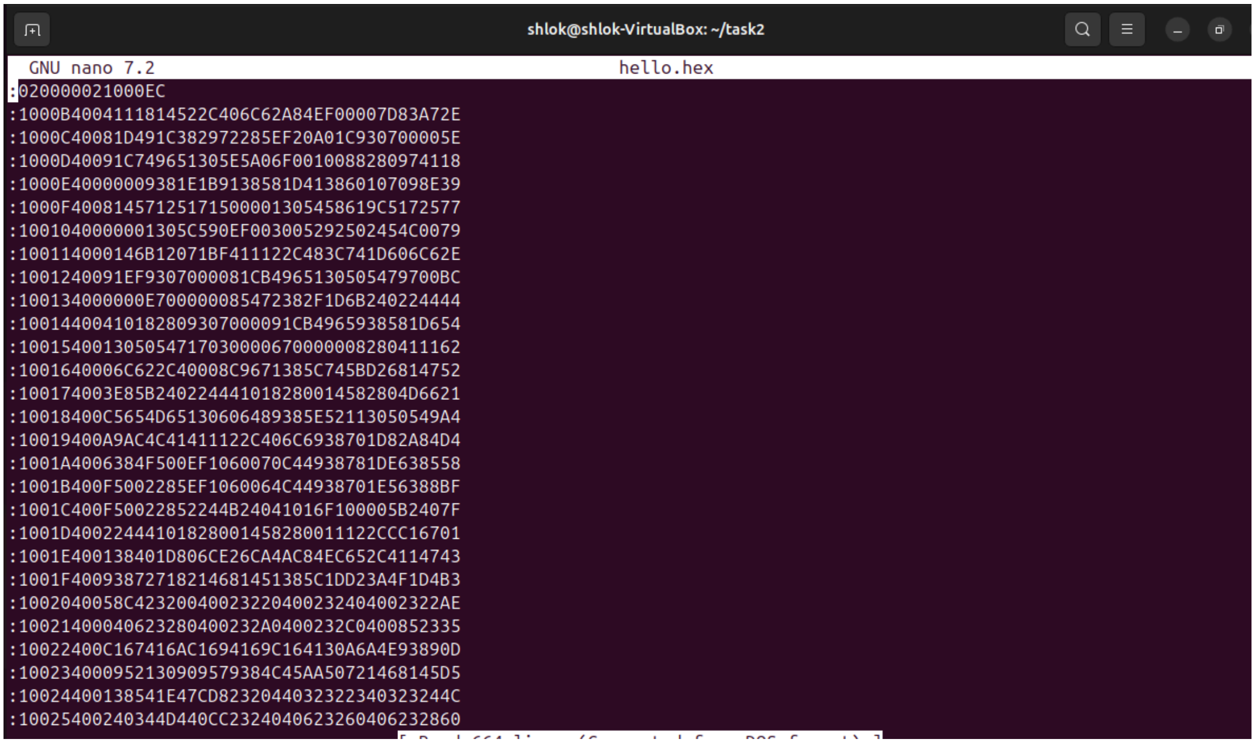
Use `hexdump` for raw binaries:

```
 riscv32-unknown-elf-objcopy -O binary hello.elf hello.bin
hexdump -C hello.bin
```

OUTPUT:



```
hello.elf:     file format elf32-littleriscv


Disassembly of section .text:

000100b4 <exit>:
   100b4:       1141 [redacted]        addi    sp,sp,-16
   100b6:       4581                    li      a1,0
   100b8:       c422                    sw      s0,8(sp)
   100ba:       c606                    sw      ra,12(sp)
   100bc:       842a                    mv      s0,a0
   100be:       7d0000ef                jal     1088e <__call_exitprocs>
   100c2:       d481a783                lw      a5,-696(gp) # 139c8 <__stdio_exit_handler>
   100c6:       c391                    beqz    a5,100ca <exit+0x16>
   100c8:       9782                    jalr    a5
   100ca:       8522                    mv      a0,s0
   100cc:       1ca020ef                jal     12296 <_exit>

000100d0 <register_fini>:
   100d0:       00000793 [redacted]    li      a5,0
   100d4:       c791                    beqz    a5,100e0 <register_fini+0x10>
   100d6:       6549                    lui     a0,0x12
   100d8:       a0e50513                addi    a0,a0,-1522 # 11a0e <__libc_fini_array>
   100dc:       0810006f                j       1095c <atexit>
   100e0:       8082                    ret

000100e2 <_start>:
```



```
  GNU nano 7.2                                         hello.hex
:020000021000EC
:1000B4004111814522C406C62A84EF00007D83A72E
:1000C40081D491C382972285EF20A01C930700005E
:1000D40091C749651305E5A06F0010088280974118
:1000E40000009381E1B9138581D413860107098E39
:1000F400814571251715000013054586I9C5172577
:100104000000001305C590EF003005292502454C0079
:1001140000146B12071BF411122C483C741D606C62E
:1001240091EF9307000081CB49651305054797 00BC
```

- Table mapping x0-x31 to zero, ra, sp, gp, tp, t0-t6, s0-s11, a0-a7.
- Calling convention summary: a0-a7 = args/returns, s-regs callee-saved, t-regs

Caller-saved.

RESPONSE:

Here’s a complete **cheat-sheet for the 32 RV32 integer registers** (x0–x31) with their **ABI names** and **calling convention roles**:

✅ RV32I Register Table

xN	ABI Name	Role	Description
x0	zero	Hardwired zero	Always 0
x1	ra	Return address	Caller sets before function call
x2	sp	Stack pointer	Points to current stack frame
x3	gp	Global pointer	Points to global data (if used)
x4	tp	Thread pointer	Used for thread-local storage
x5	t0	Temporary	Caller-saved
x6	t1	Temporary	Caller-saved
x7	t2	Temporary	Caller-saved
x8	s0/fp	Saved register / Frame ptr	Callee-saved
x9	s1	Saved register	Callee-saved

x10	a0	Function arg 0 / return val	Caller-saved
x11	a1	Function arg 1 / return val	Caller-saved
x12	a2	Function arg 2	Caller-saved
x13	a3	Function arg 3	Caller-saved
x14	a4	Function arg 4	Caller-saved
x15	a5	Function arg 5	Caller-saved
x16	a6	Function arg 6	Caller-saved
x17	a7	Function arg 7	Caller-saved
x18	s2	Saved register	Callee-saved
x19	s3	Saved register	Callee-saved
x20	s4	Saved register	Callee-saved
x21	s5	Saved register	Callee-saved
x22	s6	Saved register	Callee-saved
x23	s7	Saved register	Callee-saved

x24	s8	Saved register	Callee-saved
x25	s9	Saved register	Callee-saved
x26	s10	Saved register	Callee-saved
x27	s11	Saved register	Callee-saved
x28	t3	Temporary	Caller-saved
x29	t4	Temporary	Caller-saved
x30	t5	Temporary	Caller-saved
x31	t6	Temporary	Caller-saved

🧠 Summary: Calling Convention Roles

Category	Registers	Responsibility
Zero Reg	`zero (x0)`	Always zero
Return Addr	`ra (x1)`	Set by caller before `call`
Stack Ptr	`sp (x2)`	Maintains stack
Args/Returns	`a0–a7 (x10–x17)`	Function args + return values

Saved (callee)	`s0–s11 (x8–x9, x18–x27)`	Must be preserved by callee
Temps (caller)	`t0–t6 (x5–x7, x28–x31)`	Can be freely overwritten
Global/Thread	`gp (x3)`, `tp (x4)`	Optional; for data model

---

6 Stepping with GDB

Question

“How do I start riscv32-unknown-elf-gdb on my ELF, set a breakpoint at main, step, and inspect registers?”

Method

• Run riscv32-unknown-elf-gdb hello.elf → target sim → break main → run.

• Use info reg a0 / disassemble to watch execution.

RESPONSE:

Here’s how to **step through your RISC-V ELF binary** using `riscv32-unknown-elf-gdb`, set breakpoints, and inspect registers.

✅ Step-by-Step: Debugging with GDB

1. Start GDB with your ELF file:

```
riscv32-unknown-elf-gdb hello.elf
```

---

2. Set up the simulator target:

```
(gdb) target sim
```

This tells GDB to use its built-in RISC-V simulator (no external hardware or QEMU needed).

---

3. Set a breakpoint at `main`:

```
(gdb) break main
```

4. Run the program:

```
(gdb) run
```

It will stop at the `main` function before the first instruction.

5. Step through instructions:

```
(gdb) stepi
```

Use `stepi` to execute **one instruction** at a time, or `nexti` to step over calls.

Use `continue` to resume until the next breakpoint.

6. Inspect Registers:

```
(gdb) info registers
```

To check a specific register, like `a0`:

```
(gdb) info reg a0
```

This shows function arguments or return values in RISC-V's calling convention.

7. Disassemble Current Code:

```
(gdb) disassemble
```

This displays the current function’s disassembly so you can see which instruction you're executing.

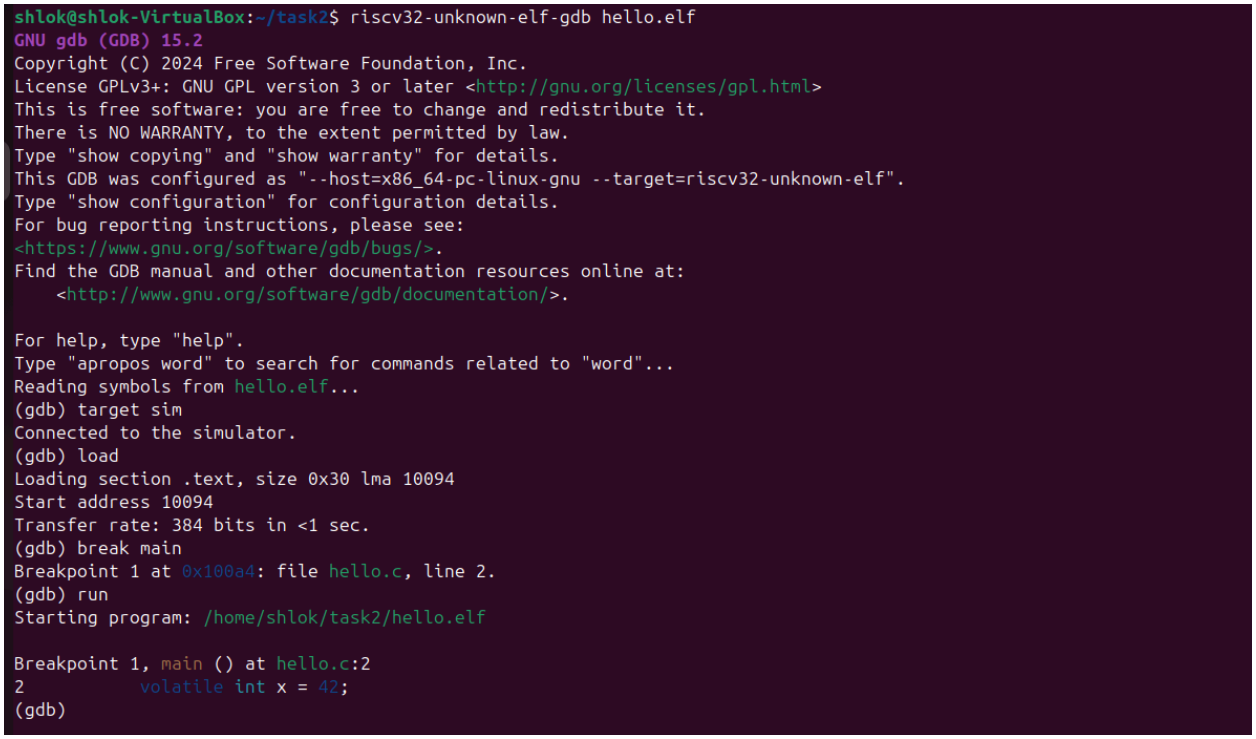
You can also disassemble a symbol:

```
(gdb) disassemble main
```

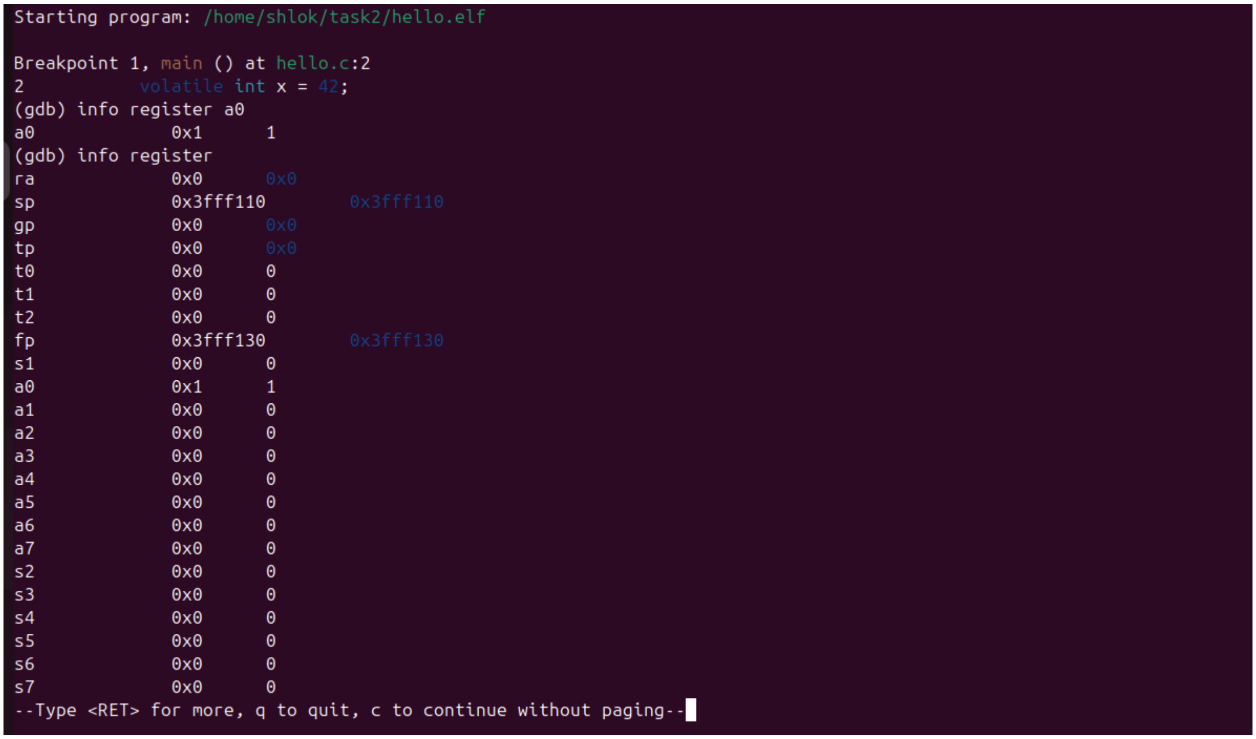
🧠 Bonus Tips

Command	Purpose
`layout asm`	TUI view of disassembly
`layout regs`	TUI view of registers
`x/4x $sp`	Examine memory at stack pointer
`bt`	Show backtrace (call stack)

OUTPUT:



```
shlok@shlok-VirtualBox:~/task2$ riscv32-unknown-elf-gdb hello.elf
GNU gdb (GDB) 15.2
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv32-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from hello.elf...
(gdb) target sim
Connected to the simulator.
(gdb) load
Loading section .text, size 0x30 lma 10094
Start address 10094
Transfer rate: 384 bits in <1 sec.
(gdb) break main
Breakpoint 1 at 0x100a4: file hello.c, line 2.
(gdb) run
Starting program: /home/shlok/task2/hello.elf

Breakpoint 1, main () at hello.c:2
2           volatile int x = 42;
(gdb)
```



```
Starting program: /home/shlok/task2/hello.elf

Breakpoint 1, main () at hello.c:2
2           volatile int x = 42;
(gdb) info register a0
a0             0x1      1
(gdb) info register
ra             0x0      0x0
sp             0x3fff110        0x3fff110
gp             0x0      0x0
tp             0x0      0x0
t0             0x0      0
t1             0x0      0
t2             0x0      0
fp             0x3fff130        0x3fff130
s1             0x0      0
a0             0x1      1
a1             0x0      0
a2             0x0      0
a3             0x0      0
a4             0x0      0
a5             0x0      0
a6             0x0      0
a7             0x0      0
s2             0x0      0
s3             0x0      0
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
--Type <RET> for more, q to quit, c to continue without paging--
```

7 Running Under an Emulator

(needed if participants don’t yet have real hardware)

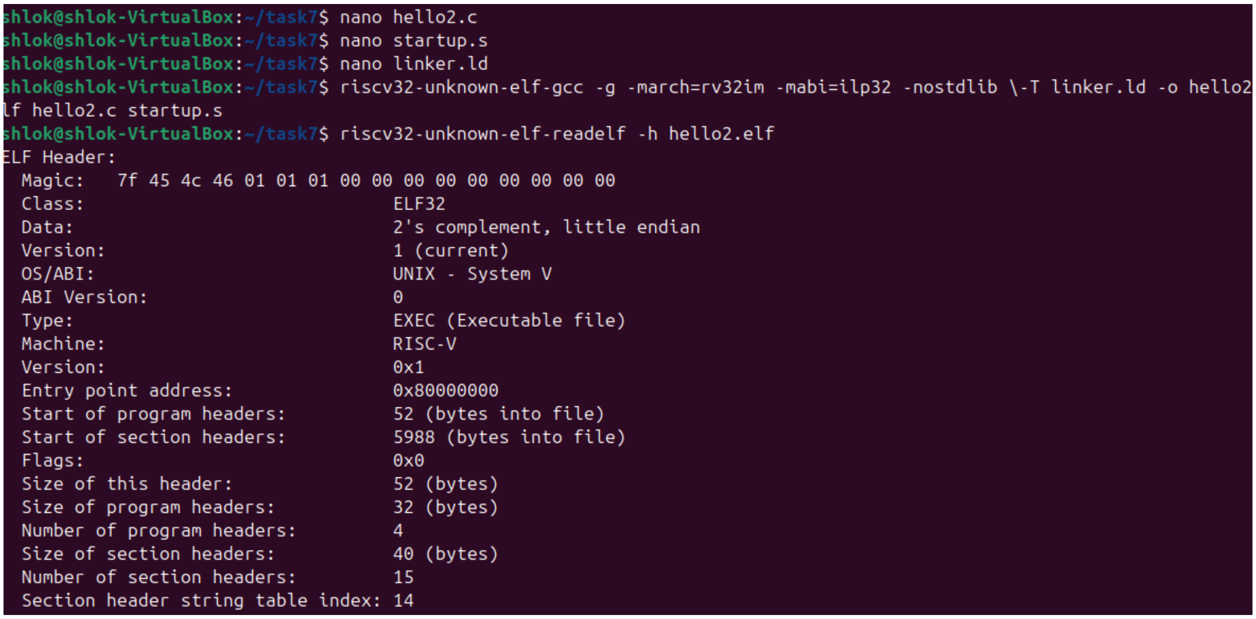
Question

“Give me spike or QEMU commands to boot my bare-metal ELF and print to the ‘UART’ console.”

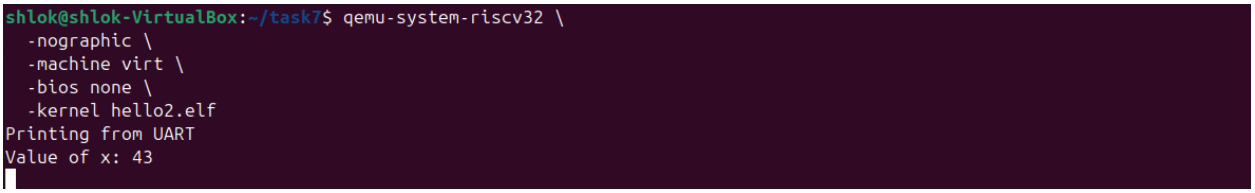
Method

- spike --isa=rv32imc pk hello.elf or
- qemu-system-riscv32 -nographic -kernel hello.elf.
- Show UART output in terminal.

RESPONSE:



```
shlok@shlok-VirtualBox:~/task7$ nano hello2.c
shlok@shlok-VirtualBox:~/task7$ nano startup.s
shlok@shlok-VirtualBox:~/task7$ nano linker.ld
shlok@shlok-VirtualBox:~/task7$ riscv32-unknown-elf-gcc -g -march=rv32im -mabi=ilp32 -nostdlib \-T linker.ld -o hello2
lf hello2.c startup.s
shlok@shlok-VirtualBox:~/task7$ riscv32-unknown-elf-readelf -h hello2.elf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x80000000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          5988 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         4
  Size of section headers:           40 (bytes)
  Number of section headers:         15
  Section header string table index: 14
```



```
shlok@shlok-VirtualBox:~/task7$ qemu-system-riscv32 \
  -nographic \
  -machine virt \
  -bios none \
  -kernel hello2.elf
Printing from UART
Value of x: 43
```

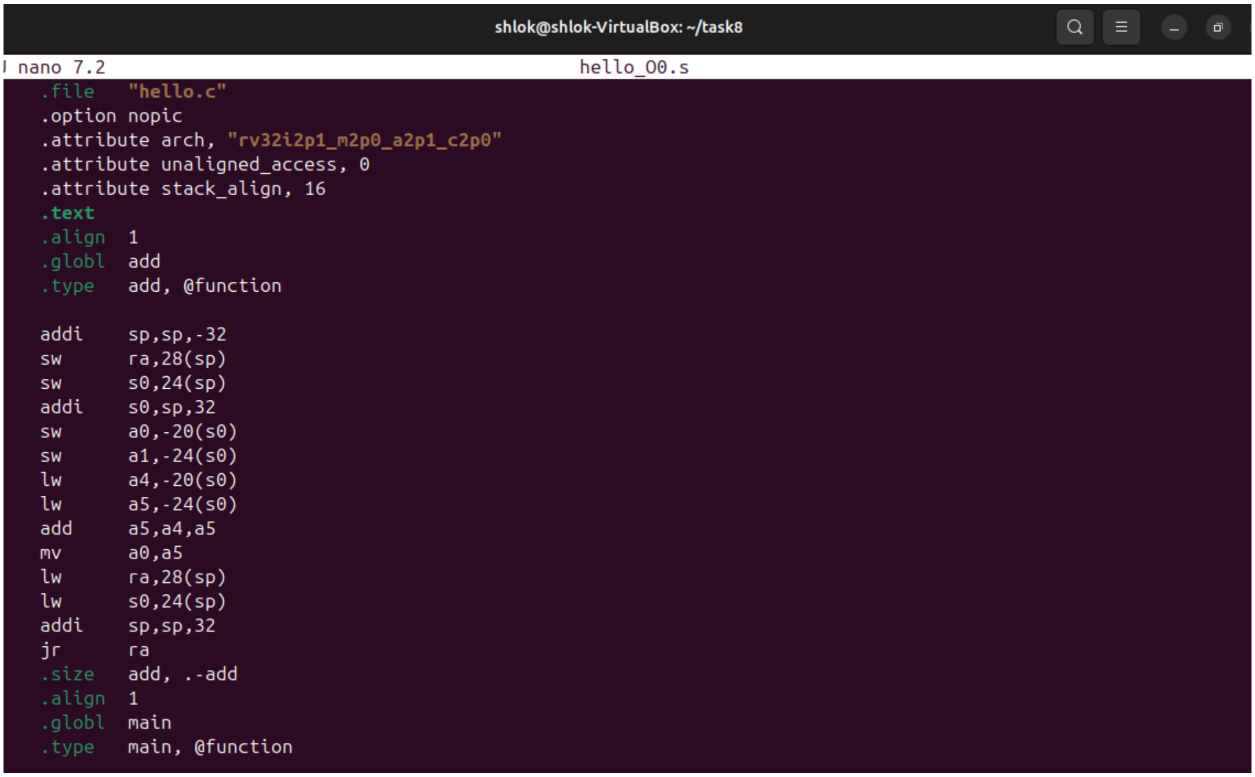
8 Exploring GCC Optimisation

Question

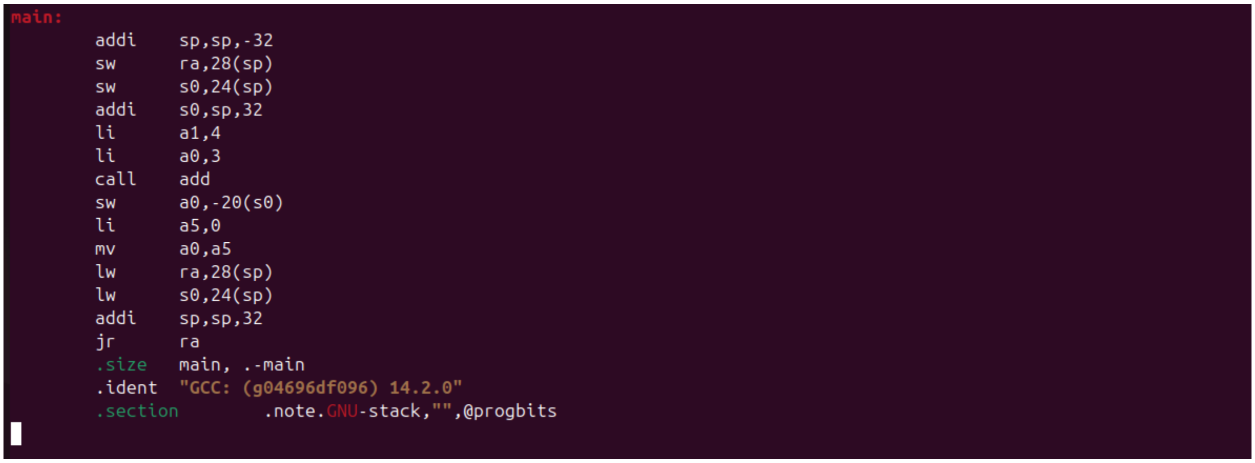
“Compile the same file with -O0 vs -O2. What differences appear in the assembly and why?”

Method

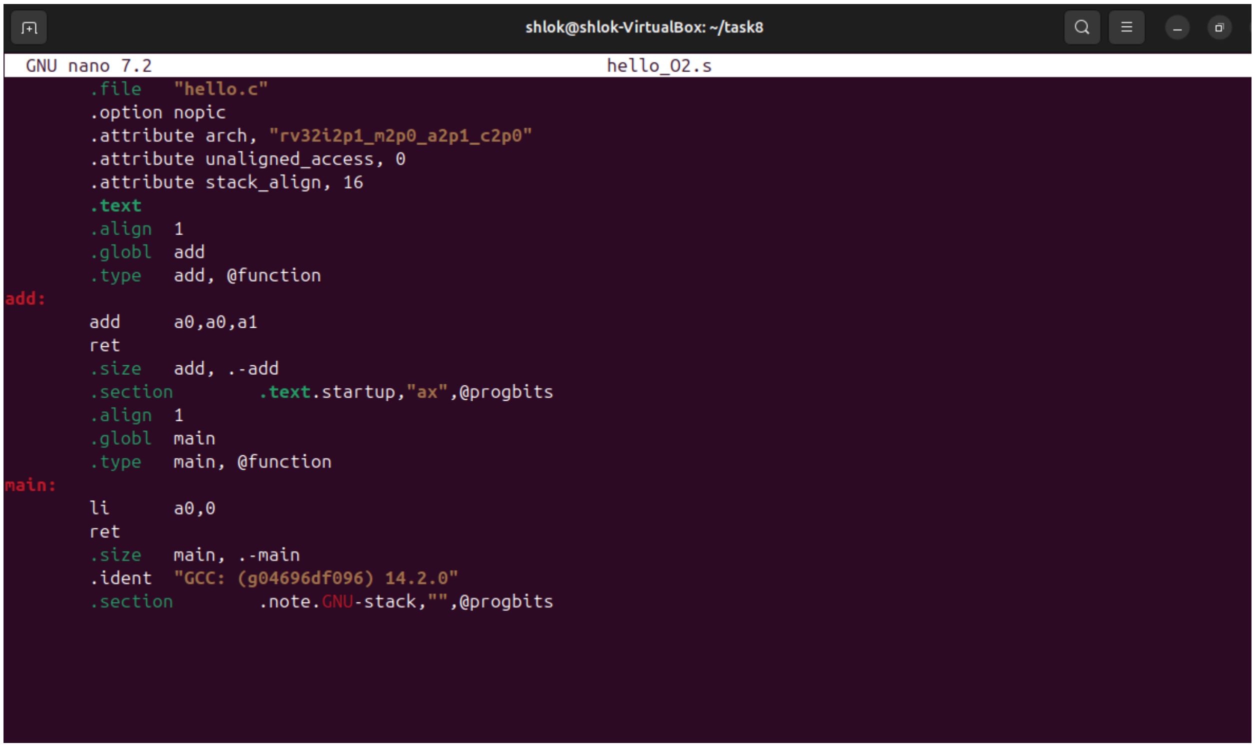
- Two .s listings side-by-side.
- ChatGPT explains dead-code elimination, register allocation, inlining.



```
 nano 7.2                                          hello_O0.s
   .file   "hello.c"
   .option nopic
   .attribute arch, "rv32i2p1_m2p0_a2p1_c2p0"
   .attribute unaligned_access, 0
   .attribute stack_align, 16
   .text
   .align  1
   .globl  add
   .type   add, @function

   addi    sp,sp,-32
   sw      ra,28(sp)
   sw      s0,24(sp)
   addi    s0,sp,32
   sw      a0,-20(s0)
   sw      a1,-24(s0)
   lw      a4,-20(s0)
   lw      a5,-24(s0)
   add     a5,a4,a5
   mv      a0,a5
   lw      ra,28(sp)
   lw      s0,24(sp)
   addi    sp,sp,32
   jr      ra
   .size   add, .-add
   .align  1
   .globl  main
   .type   main, @function
```



```
main:
        addi    sp,sp,-32
        sw      ra,28(sp)
        sw      s0,24(sp)
        addi    s0,sp,32
        li      a1,4
        li      a0,3
        call    add
        sw      a0,-20(s0)
        li      a5,0
        mv      a0,a5
        lw      ra,28(sp)
        lw      s0,24(sp)
        addi    sp,sp,32
        jr      ra
        .size   main, .-main
        .ident  "GCC: (g04696df096) 14.2.0"
        .section        .note.GNU-stack,"",@progbits
```



```
shlok@shlok-VirtualBox: ~/task8
  GNU nano 7.2                                         hello_O2.s
        .file   "hello.c"
        .option nopic
        .attribute arch, "rv32i2p1_m2p0_a2p1_c2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  add
        .type   add, @function
add:
        add     a0,a0,a1
        ret
        .size   add, .-add
        .section        .text.startup,"ax",@progbits
        .align  1
        .globl  main
        .type   main, @function
main:
        li      a0,0
        ret
        .size   main, .-main
        .ident  "GCC: (g04696df096) 14.2.0"
        .section        .note.GNU-stack,"",@progbits
```

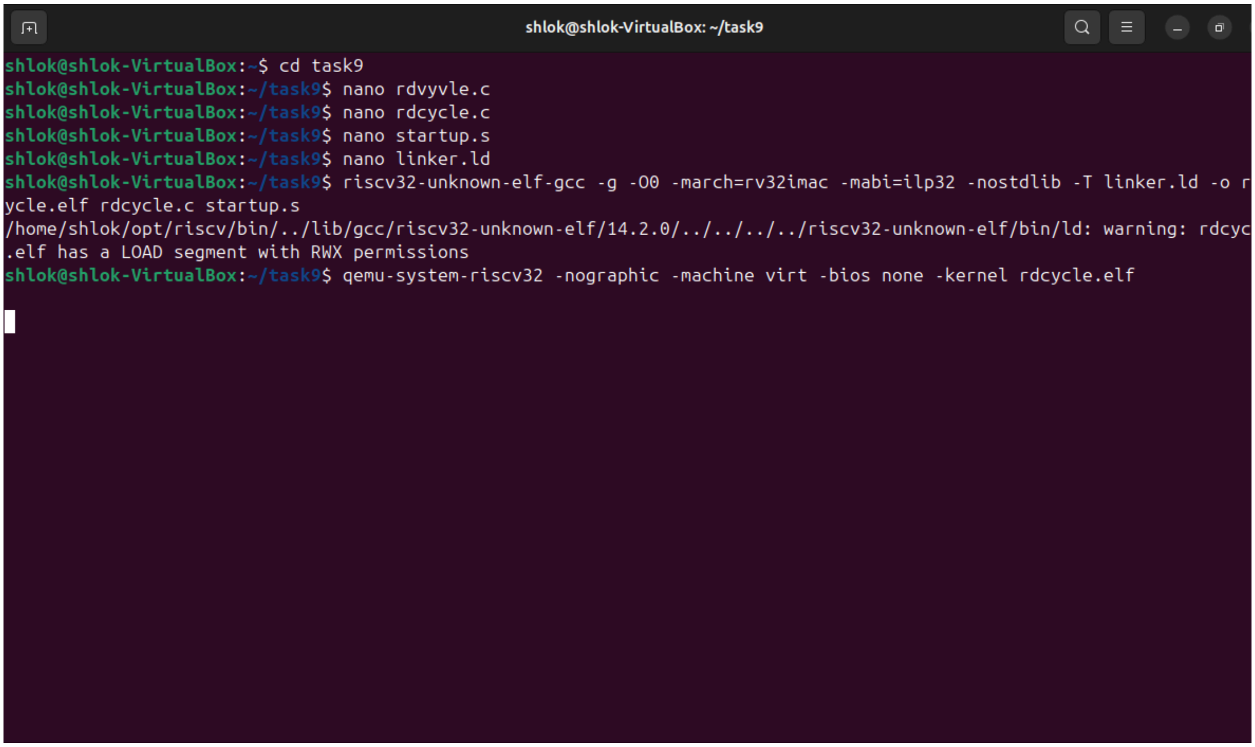
9 Inline Assembly Basics

Question

“Write a C function that returns the cycle counter by reading CSR 0xC00 using inline asm; explain each constraint.”

Method

- Provide static inline uint32_t rdcycle(void) { uint32_t c; asm volatile ("csrr %0, cycle" : "=r"(c)); return c; }.
- Break down "=r"(c) and volatile.



```
shlok@shlok-VirtualBox:~$ cd task9
shlok@shlok-VirtualBox:~/task9$ nano rdvyvle.c
shlok@shlok-VirtualBox:~/task9$ nano rdcycle.c
shlok@shlok-VirtualBox:~/task9$ nano startup.s
shlok@shlok-VirtualBox:~/task9$ nano linker.ld
shlok@shlok-VirtualBox:~/task9$ riscv32-unknown-elf-gcc -g -O0 -march=rv32imac -mabi=ilp32 -nostdlib -T linker.ld -o r
ycle.elf rdcycle.c startup.s
/home/shlok/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/14.2.0/../../../../riscv32-unknown-elf/bin/ld: warning: rdcyc
.elf has a LOAD segment with RWX permissions
shlok@shlok-VirtualBox:~/task9$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel rdcycle.elf

```

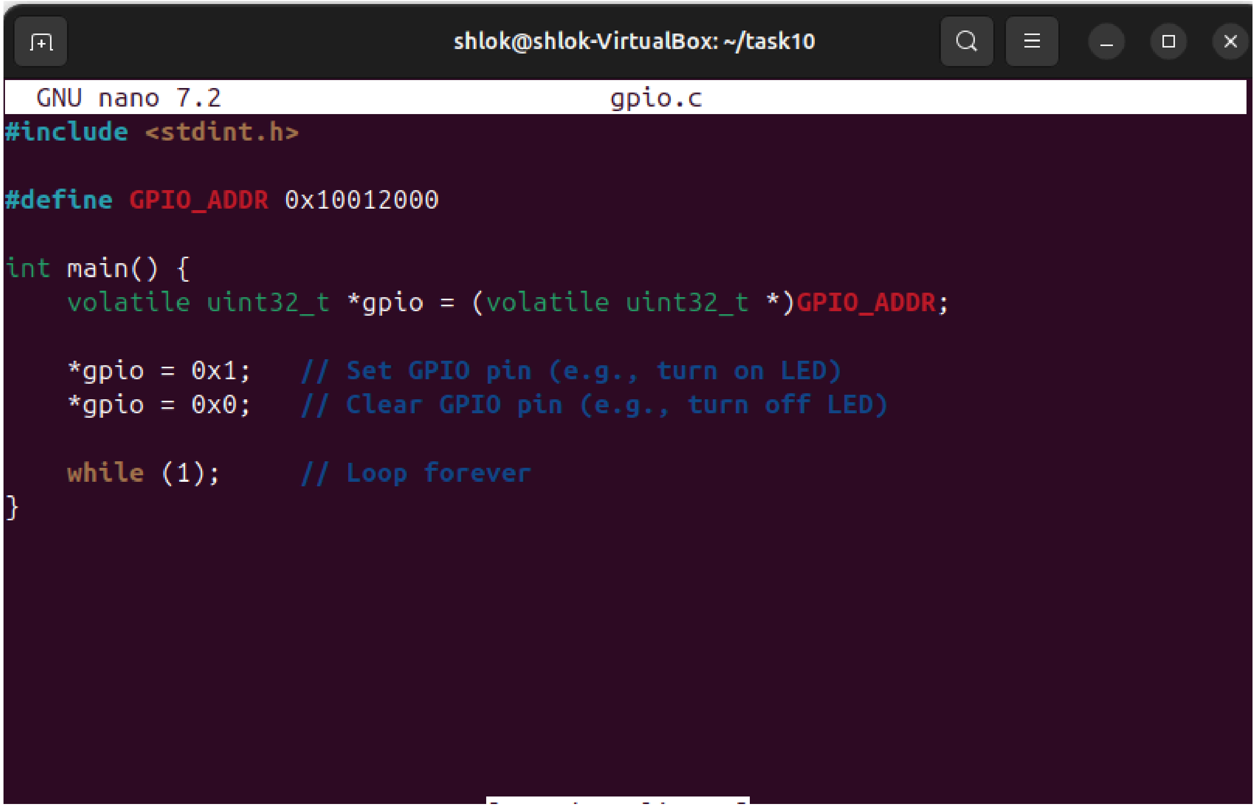
10 Memory-Mapped I/O Demo

Question

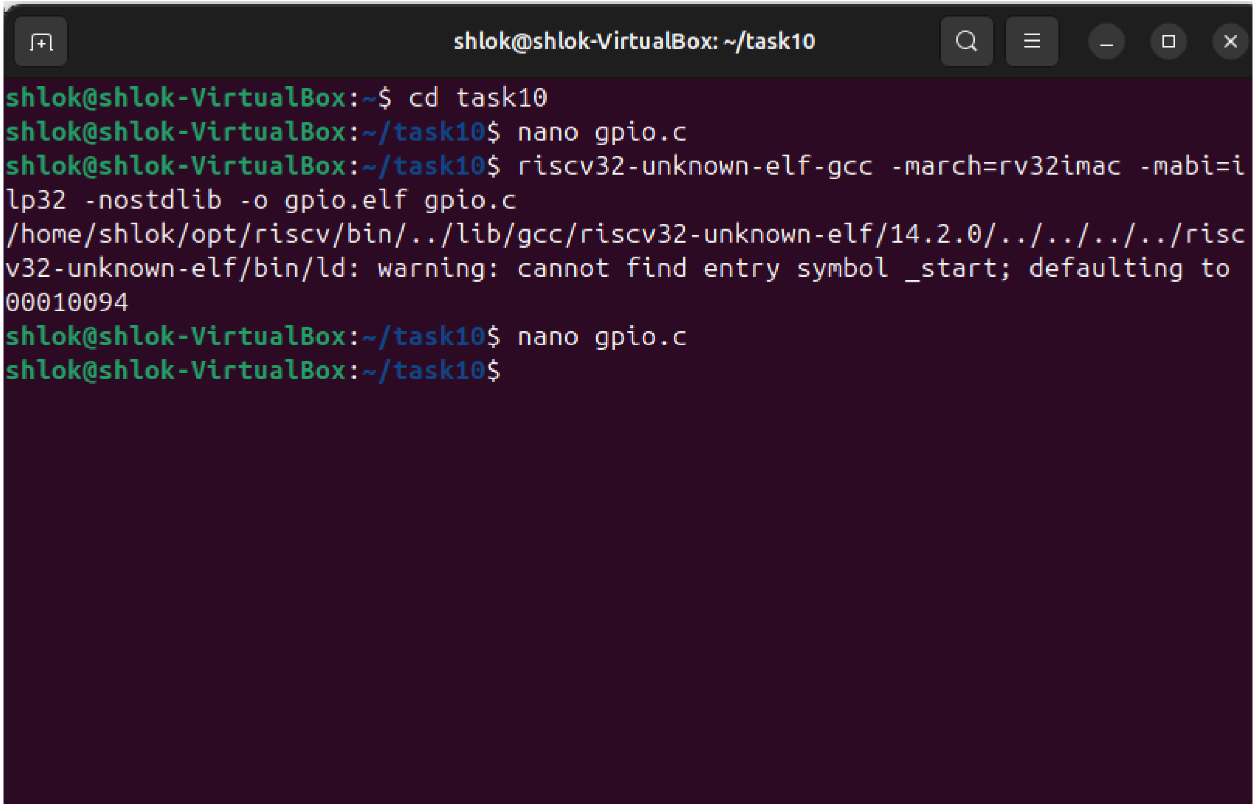
“Show a bare-metal C snippet to toggle a GPIO register located at 0x10012000. How do I prevent the compiler from optimising the store away?”

Method

- Use volatile uint32_t *gpio = (uint32_t*)0x10012000; *gpio = 0x1;.
- Discuss the volatile keyword and alignment.



```
 GNU nano 7.2                          gpio.c
#include <stdint.h>

#define GPIO_ADDR 0x10012000

int main() {
    volatile uint32_t *gpio = (volatile uint32_t *)GPIO_ADDR;

    *gpio = 0x1;    // Set GPIO pin (e.g., turn on LED)
    *gpio = 0x0;    // Clear GPIO pin (e.g., turn off LED)

    while (1);      // Loop forever
}
```



```
shlok@shlok-VirtualBox:~$ cd task10
shlok@shlok-VirtualBox:~/task10$ nano gpio.c
shlok@shlok-VirtualBox:~/task10$ riscv32-unknown-elf-gcc -march=rv32imac -mabi=i
lp32 -nostdlib -o gpio.elf gpio.c
/home/shlok/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/14.2.0/../../../../risc
v32-unknown-elf/bin/ld: warning: cannot find entry symbol _start; defaulting to
00010094
shlok@shlok-VirtualBox:~/task10$ nano gpio.c
shlok@shlok-VirtualBox:~/task10$
```

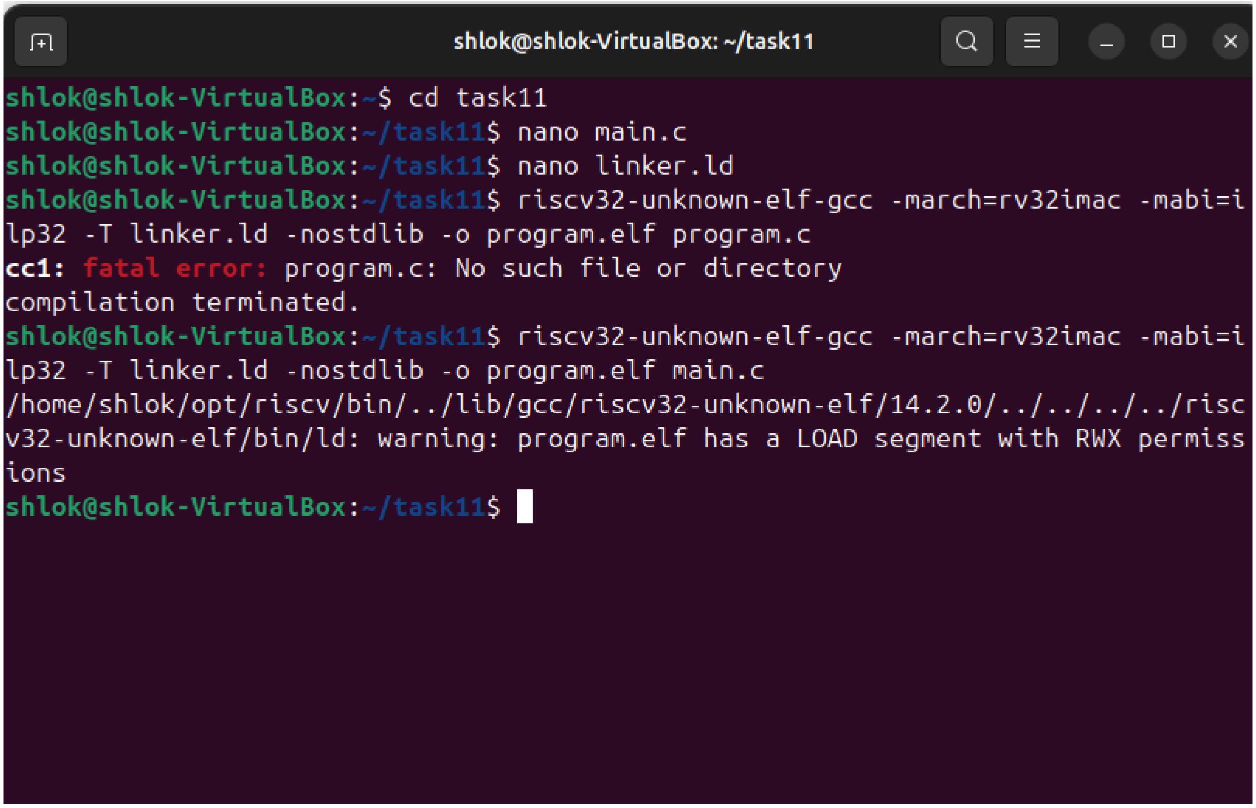
11 Linker Script 101

Question

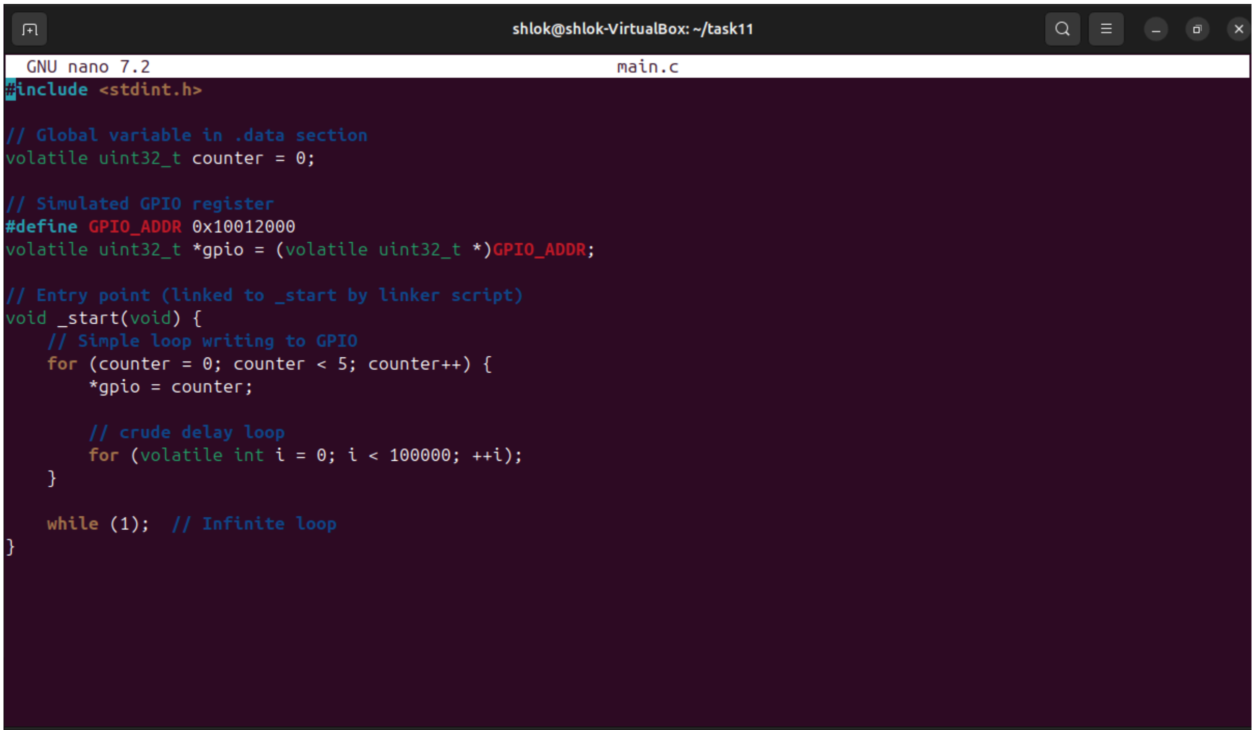
“Provide a minimal linker script that places .text at 0x00000000 and .data at 0x10000000 for RV32IMC.”

Method

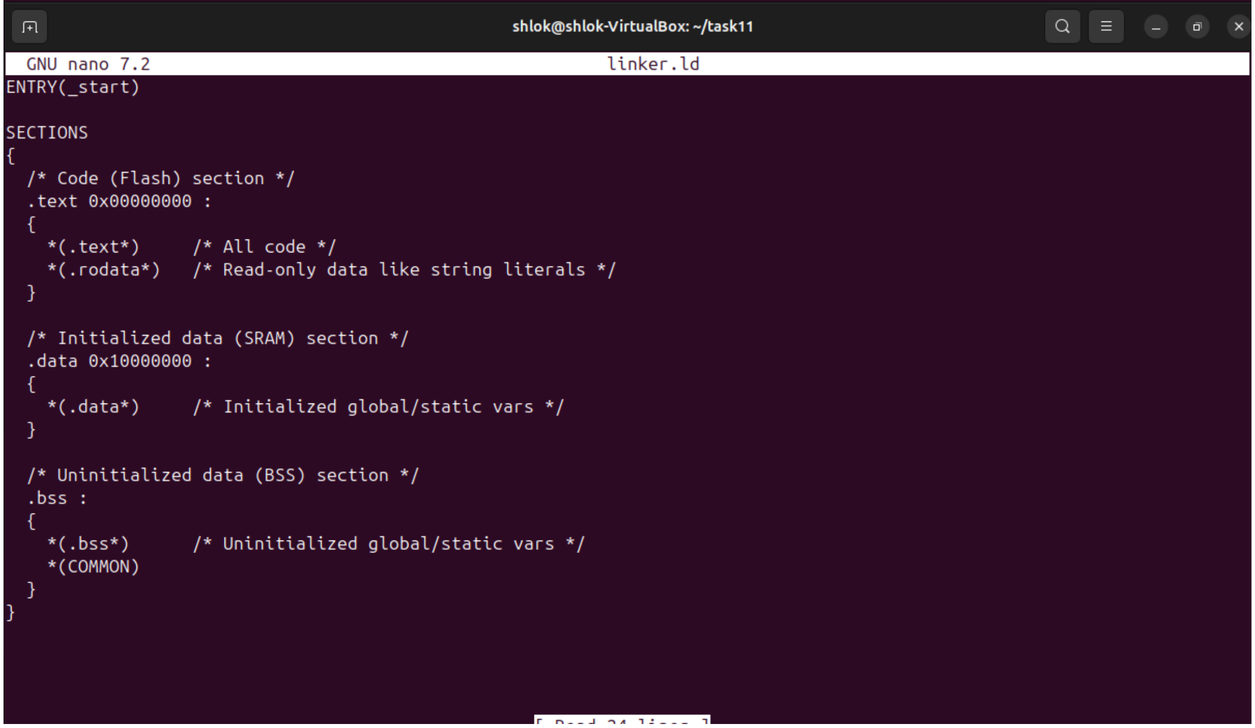
- Skeleton SECTIONS { .text 0x0 : { *(.text*) } .data 0x10000000 : { *(.data*) } }.
- Explain why Flash vs SRAM addresses differ



```
shlok@shlok-VirtualBox:~$ cd task11
shlok@shlok-VirtualBox:~/task11$ nano main.c
shlok@shlok-VirtualBox:~/task11$ nano linker.ld
shlok@shlok-VirtualBox:~/task11$ riscv32-unknown-elf-gcc -march=rv32imac -mabi=i
lp32 -T linker.ld -nostdlib -o program.elf program.c
cc1: fatal error: program.c: No such file or directory
compilation terminated.
shlok@shlok-VirtualBox:~/task11$ riscv32-unknown-elf-gcc -march=rv32imac -mabi=i
lp32 -T linker.ld -nostdlib -o program.elf main.c
/home/shlok/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/14.2.0/../../../../risc
v32-unknown-elf/bin/ld: warning: program.elf has a LOAD segment with RWX permiss
ions
shlok@shlok-VirtualBox:~/task11$ 
```



```c
#include <stdint.h>

// Global variable in .data section
volatile uint32_t counter = 0;

// Simulated GPIO register
#define GPIO_ADDR 0x10012000
volatile uint32_t *gpio = (volatile uint32_t *)GPIO_ADDR;

// Entry point (linked to _start by linker script)
void _start(void) {
    // Simple loop writing to GPIO
    for (counter = 0; counter < 5; counter++) {
        *gpio = counter;

        // crude delay loop
        for (volatile int i = 0; i < 100000; ++i);
    }

    while (1);  // Infinite loop
}
```



```
ENTRY(_start)

SECTIONS
{
  /* Code (Flash) section */
  .text 0x00000000 :
  {
    *(.text*)     /* All code */
    *(.rodata*)   /* Read-only data like string literals */
  }

  /* Initialized data (SRAM) section */
  .data 0x10000000 :
  {
    *(.data*)     /* Initialized global/static vars */
  }

  /* Uninitialized data (BSS) section */
  .bss :
  {
    *(.bss*)      /* Uninitialized global/static vars */
    *(COMMON)
  }
}
```

12 Start-up Code & crt0

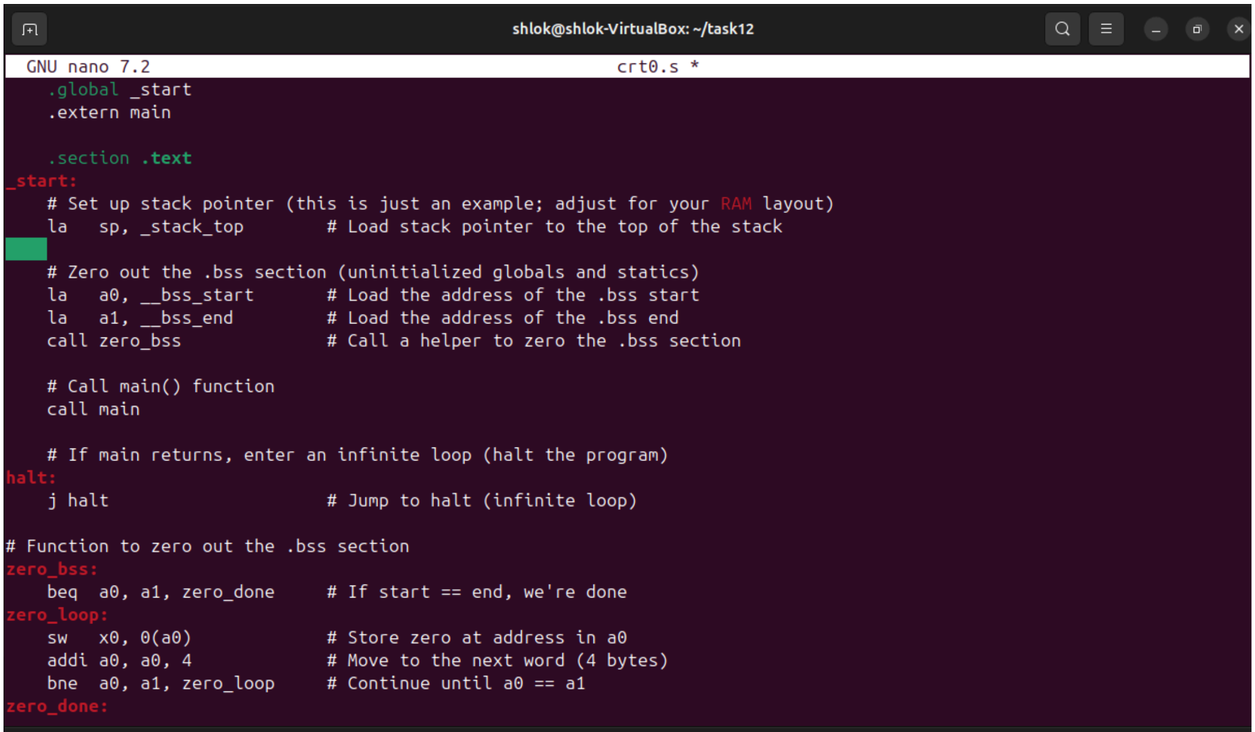
Question

“What does crt0.S typically do in a bare-metal RISC-V program and where do I get one?”

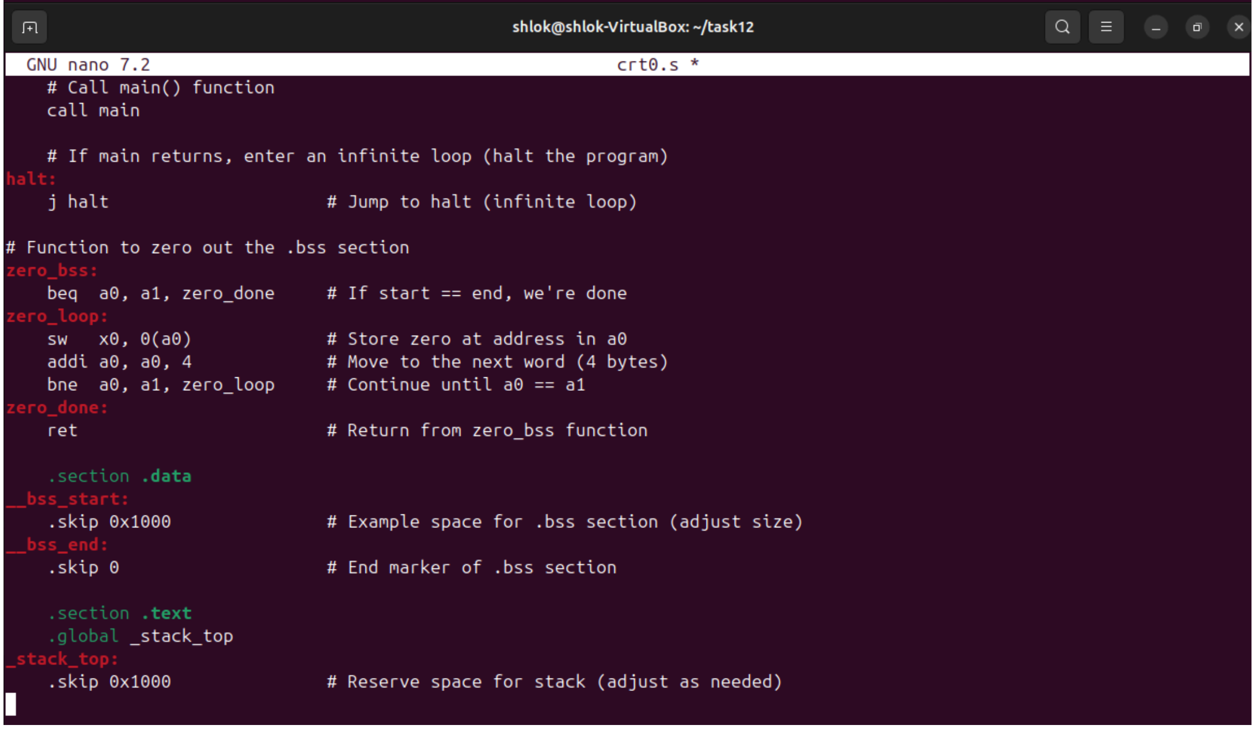
Answer Highlights

- Sets up stack pointer, zeroes .bss, calls main, enters infinite loop.

- Point to device-specific examples or newlib.



```
  GNU nano 7.2                                        crt0.s *
    .global _start
    .extern main

    .section .text
_start:
    # Set up stack pointer (this is just an example; adjust for your RAM layout)
    la   sp, _stack_top        # Load stack pointer to the top of the stack

    # Zero out the .bss section (uninitialized globals and statics)
    la   a0, __bss_start       # Load the address of the .bss start
    la   a1, __bss_end         # Load the address of the .bss end
    call zero_bss              # Call a helper to zero the .bss section

    # Call main() function
    call main

    # If main returns, enter an infinite loop (halt the program)
halt:
    j halt                     # Jump to halt (infinite loop)

# Function to zero out the .bss section
zero_bss:
    beq  a0, a1, zero_done     # If start == end, we're done
zero_loop:
    sw   x0, 0(a0)             # Store zero at address in a0
    addi a0, a0, 4             # Move to the next word (4 bytes)
    bne  a0, a1, zero_loop     # Continue until a0 == a1
zero_done:
```



```
  GNU nano 7.2                                        crt0.s *
    # Call main() function
    call main

    # If main returns, enter an infinite loop (halt the program)
halt:
    j halt                     # Jump to halt (infinite loop)

# Function to zero out the .bss section
zero_bss:
    beq  a0, a1, zero_done     # If start == end, we're done
zero_loop:
    sw   x0, 0(a0)             # Store zero at address in a0
    addi a0, a0, 4             # Move to the next word (4 bytes)
    bne  a0, a1, zero_loop     # Continue until a0 == a1
zero_done:
    ret                        # Return from zero_bss function

    .section .data
__bss_start:
    .skip 0x1000               # Example space for .bss section (adjust size)
__bss_end:
    .skip 0                    # End marker of .bss section

    .section .text
    .global _stack_top
_stack_top:
    .skip 0x1000               # Reserve space for stack (adjust as needed)

```

13 Interrupt Primer

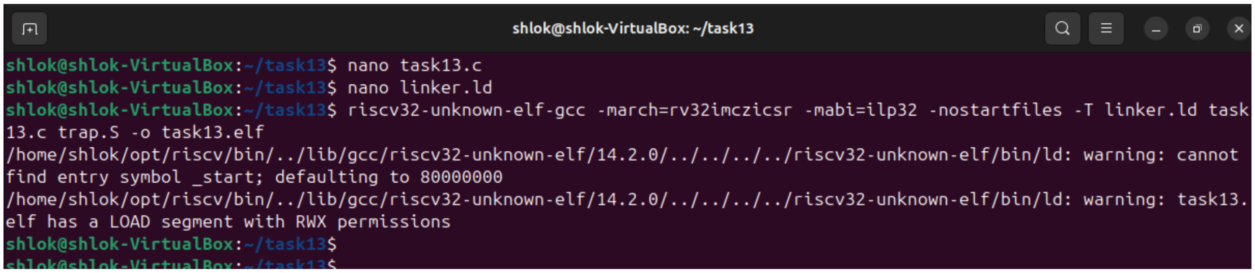
Question

“Demonstrate how to enable the machine-timer interrupt (MTIP) and write a simple handler in C/asm.”

Method

• Write to mtimecmp, set mie, poll mstatus.

• Use __attribute__((interrupt)) or naked ISR pattern.



```
shlok@shlok-VirtualBox:~/task13$ nano task13.c
shlok@shlok-VirtualBox:~/task13$ nano linker.ld
shlok@shlok-VirtualBox:~/task13$ riscv32-unknown-elf-gcc -march=rv32imczicsr -mabi=ilp32 -nostartfiles -T linker.ld task
13.c trap.S -o task13.elf
/home/shlok/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/14.2.0/../../../../riscv32-unknown-elf/bin/ld: warning: cannot
find entry symbol _start; defaulting to 80000000
/home/shlok/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/14.2.0/../../../../riscv32-unknown-elf/bin/ld: warning: task13.
elf has a LOAD segment with RWX permissions
shlok@shlok-VirtualBox:~/task13$
shlok@shlok-VirtualBox:~/task13$
```

14. Explain the ‘A’ (atomic) extension in rv32imac. What instructions are added and why are they useful?

Ans. The ‘A’ (Atomic) extension in RV32IMAC introduces instructions that support atomic read-modify-write operations. These operations are essential in systems where multiple execution threads or processor cores might access and modify the same memory location at the same time. Without atomic instructions, such shared access could lead to race conditions, incorrect program behavior, or system crashes.

The extension adds several key instructions:

- `lr.w` (Load Reserved): Loads a word from memory and marks it for exclusive access.
- `sc.w` (Store Conditional): Attempts to store a word to memory only if no other core or thread has modified it since the corresponding `lr.w`.
- `amoswap.w`: Atomically swaps a value in a register with a value in memory.
- `amoadd.w`: Atomically adds a value to a memory location.
- `amoor.w`, `amoand.w`, `amoxor.w`: Perform atomic bitwise OR, AND, and XOR on memory.

`amomin.w`, `amomax.w`, `amominu.w`, `amomaxu.w`: Perform atomic min/max operations (signed and unsigned) on memory.

These instructions are particularly useful for implementing synchronization mechanisms such as spinlocks, mutexes, and semaphores. They allow multiple cores or threads to coordinate without the need for disabling interrupts or using complex software-based locking methods. This is important in both operating system kernels and user-level applications that require thread-safe access to shared data.

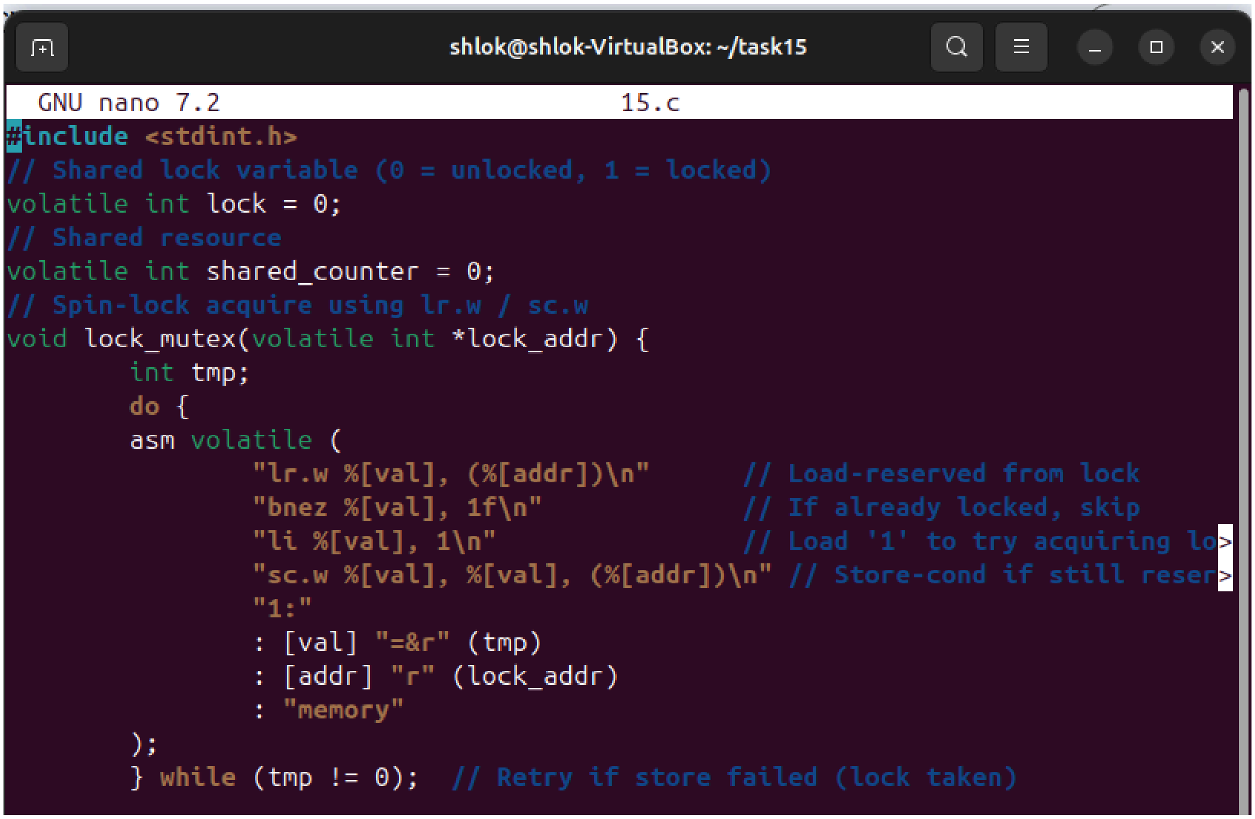
15 Atomic Test Program

Question

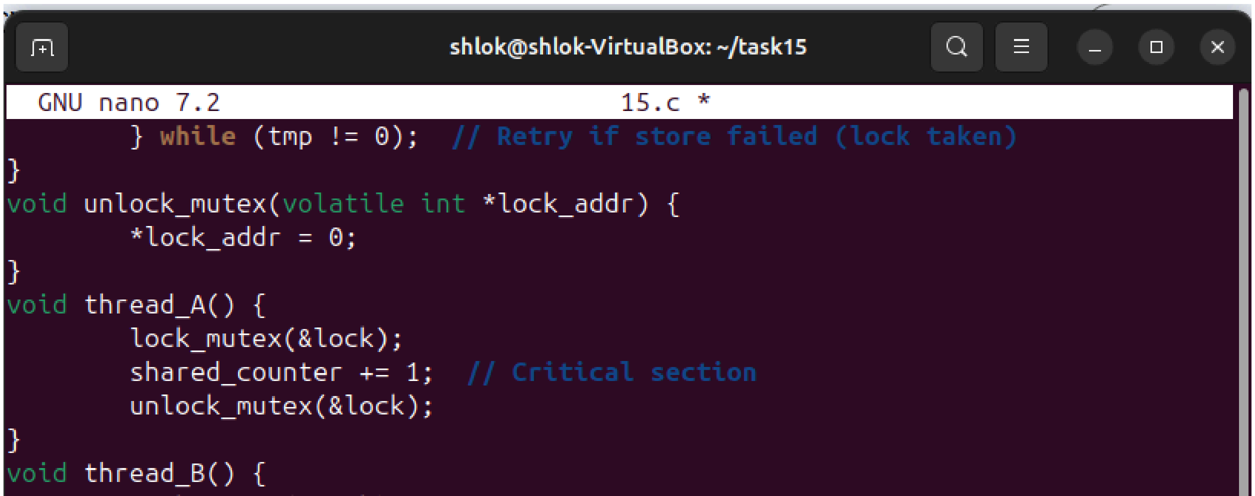
“Provide a two-thread mutex example (pseudo-threads in main) using lr/sc on RV32.”

Method

- Spin-lock implementation in C with inline asm fallback.



```
  GNU nano 7.2                          15.c
#include <stdint.h>
// Shared lock variable (0 = unlocked, 1 = locked)
volatile int lock = 0;
// Shared resource
volatile int shared_counter = 0;
// Spin-lock acquire using lr.w / sc.w
void lock_mutex(volatile int *lock_addr) {
        int tmp;
        do {
        asm volatile (
                "lr.w %[val], (%[addr])\n"      // Load-reserved from lock
                "bnez %[val], 1f\n"             // If already locked, skip
                "li %[val], 1\n"                // Load '1' to try acquiring lo>
                "sc.w %[val], %[val], (%[addr])\n" // Store-cond if still reser>
                "1:"
                : [val] "=&r" (tmp)
                : [addr] "r" (lock_addr)
                : "memory"
        );
        } while (tmp != 0);  // Retry if store failed (lock taken)
```



```
  GNU nano 7.2                          15.c *
        } while (tmp != 0);  // Retry if store failed (lock taken)
}
void unlock_mutex(volatile int *lock_addr) {
        *lock_addr = 0;
}
void thread_A() {
        lock_mutex(&lock);
        shared_counter += 1;  // Critical section
        unlock_mutex(&lock);
}
void thread_B() {
```

lr.w (Load-Reserved) reads the lock. sc.w (Store-Conditional) writes only if no one else wrote since lr.w. If sc.w fails (returns non-zero), the loop retries — this is the spin-lock behavior. thread_A and thread_B are simulated threads using critical sections. This example is simple and suitable for bare-metal RV32 systems without an operating system or threads

To use `printf` from Newlib without an OS (i.e., in a bare-metal RISC-V environment), you need to **retarget the `_write` syscall** so that it redirects output to a memory-mapped UART. Here's how to do it.

---

16 Using Newlib printf Without an OS

Question

“How do I retarget _write so that printf sends bytes to my memory-mapped UART?”

Answer Outline

- Implement _write(int fd, char* buf, int len) that loops over bytes to UART_TX.

- Re-link with -nostartfiles + custom syscalls.c.

✅ Step-by-Step: Retargeting `_write` for Bare-Metal UART Output

🔧 Step 1: Implement `_write` in `syscalls.c`

Create a `syscalls.c` file with the following minimal implementation:

```
#include <unistd.h>

#include <stdint.h>



#define UART0_TX 0x10000000  // Adjust address as per your platform



volatile char* const uart = (char*)UART0_TX;



int _write(int fd, const char* buf, int len) {
    for (int i = 0; i < len; i++) {
        uart[0] = buf[i];  // Write each byte to UART TX
    }
    return len;
}
```

> This sends each character to your memory-mapped UART (typically a UART TX register).

🛠️ Step 2: Recompile and Link with Newlib (Bare-Metal)

Use `-nostartfiles` to skip Newlib's default `crt0` and instead provide your own `syscalls.c`.

```
riscv32-unknown-elf-gcc -nostartfiles -march=rv32imc -mabi=ilp32 \
  -T linker.ld -o hello.elf hello.c syscalls.c
```

You must also define `_start` somewhere in your code to replace the OS's entry point.

📦 Linker Script Consideration

Ensure your `linker.ld` defines correct sections like `.text`, `.data`, and a stack:

```
SECTIONS {
  . = 0x80000000;
  .text : { *(.text*) }
  .data : { *(.data*) }
  .bss  : { *(.bss*) }
  .stack (NOLOAD) : {
    . = ALIGN(8);
    _stack_top = .;
    . = . + 0x1000;
  }
}
```

🧠 Notes

- `fd` is ignored in this example (i.e., it treats all output as stdout).
- Don’t forget to declare `extern int _write(...)` if you call it manually.
- To use more `printf`-based Newlib features (e.g. `scanf`), you'd need to also implement `_read`, `_close`, `_lseek`, `_fstat`, etc.

---

17 Endianness & Struct Packing

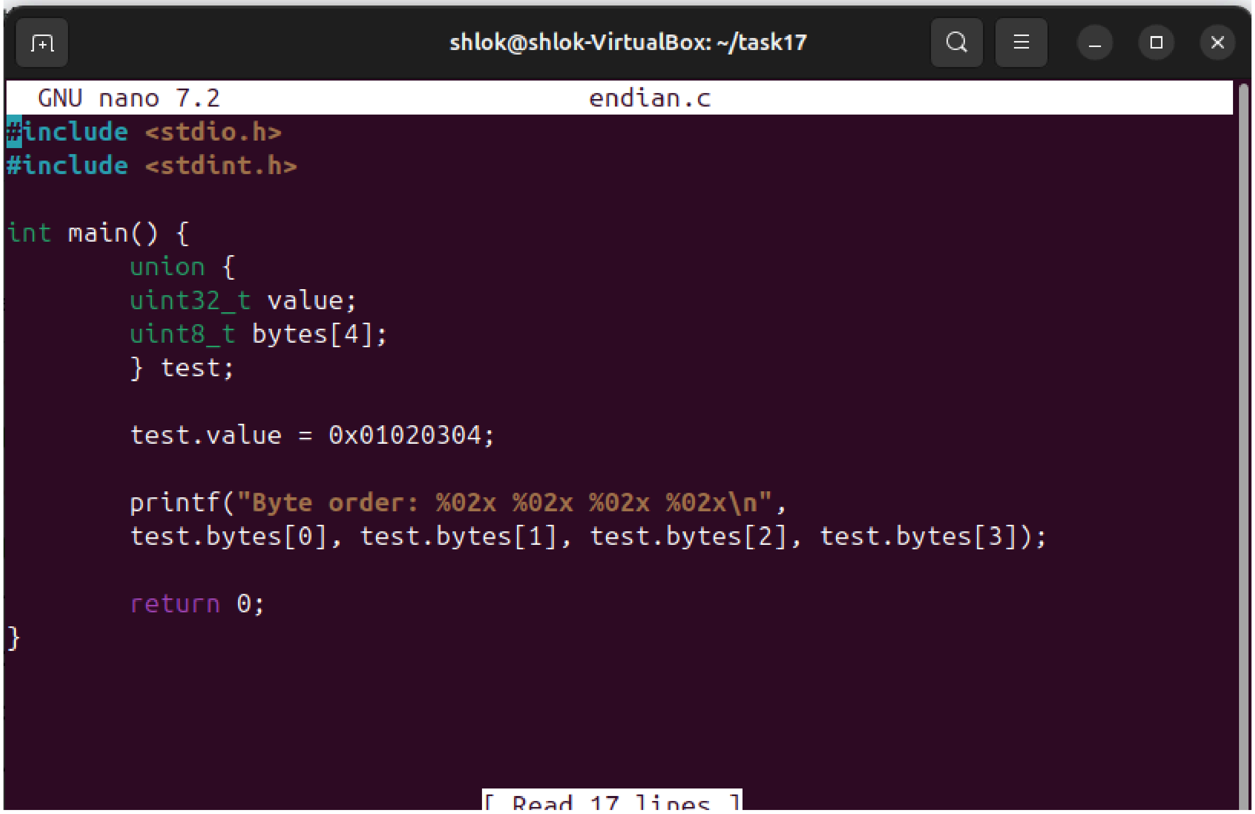
Question

“Is RV32 little-endian by default? Show me how to verify byte ordering with a union trick in C.”

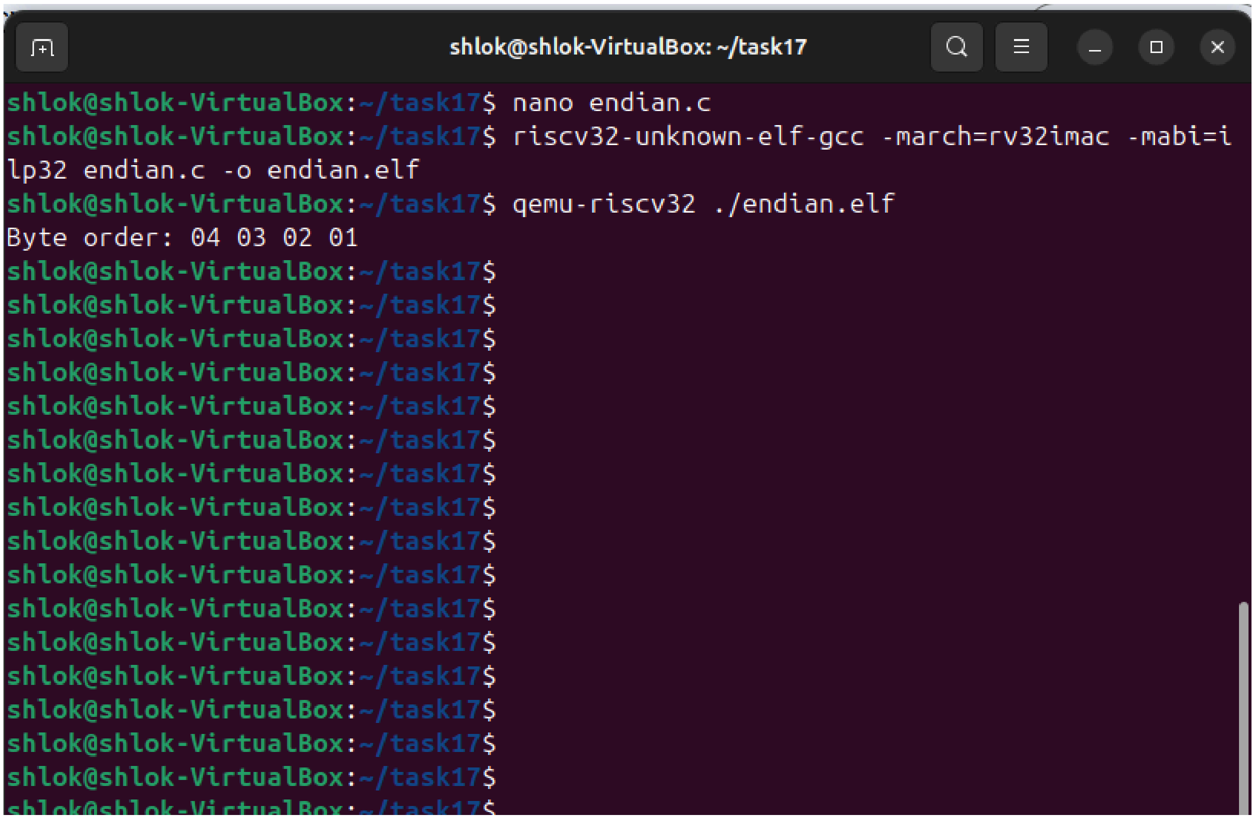
Method

• Union of uint32_t and uint8_t[4], store 0x01020304, print bytes.

Ans.



```
  GNU nano 7.2                        endian.c
#include <stdio.h>
#include <stdint.h>

int main() {
        union {
        uint32_t value;
        uint8_t bytes[4];
        } test;

        test.value = 0x01020304;

        printf("Byte order: %02x %02x %02x %02x\n",
        test.bytes[0], test.bytes[1], test.bytes[2], test.bytes[3]);

        return 0;
}



                               [ Read 17 lines ]
```



```
shlok@shlok-VirtualBox:~/task17$ nano endian.c
shlok@shlok-VirtualBox:~/task17$ riscv32-unknown-elf-gcc -march=rv32imac -mabi=i
lp32 endian.c -o endian.elf
shlok@shlok-VirtualBox:~/task17$ qemu-riscv32 ./endian.elf
Byte order: 04 03 02 01
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
shlok@shlok-VirtualBox:~/task17$
```