

*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?*

**AI Response:**

Here's a clear, step-by-step guide on how to **unpack the xPack RISC-V toolchain, add it to your PATH, and verify that gcc, objdump, and gdb work properly**:

## 1. Download the Toolchain

(Use the latest version; for example:)

```
wget https://github.com/xpack-dev-tools/riscv-none-elf-gcc-xpack/releases/download/v14.2.0-3/xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64.tar.gz
```

## 2. Unpack the Archive

```
tar -xvzf xpack-riscv-none-elf-gcc-14.2.0-3-linux-x64.tar.gz
```

This creates a folder named something like xpack-riscv-none-elf-gcc-14.2.0-3.

## 3. Add the Toolchain to Your PATH (Temporarily)

Run this command in your terminal to add the toolchain binaries to your current session's PATH:

```
export PATH=$PWD/xpack-riscv-none-elf-gcc-14.2.0-3/bin:$PATH
```

- Here, \$PWD is your current directory where the extracted folder is located.
- This makes commands like riscv-none-elf-gcc accessible right away.

## 4. Check if Binaries Work

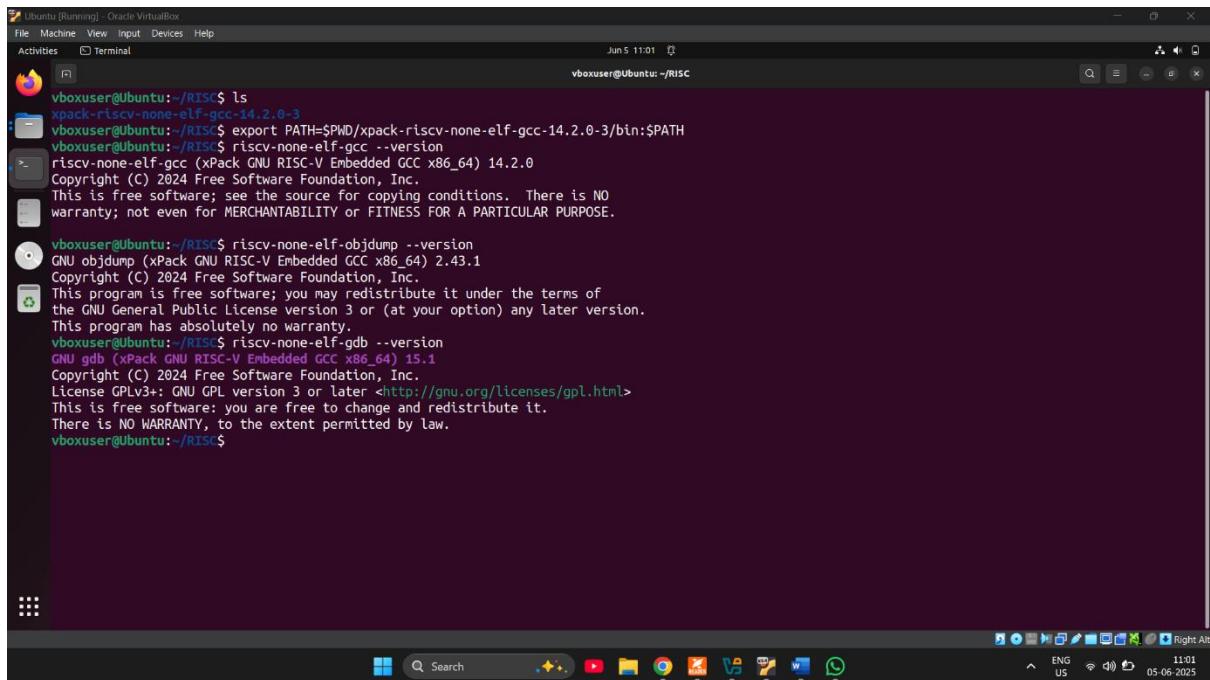
```
riscv-none-elf-gcc --version
```

```
riscv-none-elf-objdump --version
```

```
riscv-none-elf-gdb --version
```

Each should output version information confirming they work.

## Output:



```
vboxuser@Ubuntu:~/RISC$ ls
xpack-riscv-none-elf-gcc-14.2.0-3
vboxuser@Ubuntu:~/RISC$ export PATH=$PWD/xpack-riscv-none-elf-gcc-14.2.0-3/bin:$PATH
vboxuser@Ubuntu:~/RISC$ riscv-none-elf-gcc --version
riscv-none-elf-gcc (xPack GNU RISC-V Embedded GCC x86_64) 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.

vboxuser@Ubuntu:~/RISC$ riscv-none-elf-objdump --version
GNU objdump (xPack GNU RISC-V Embedded GCC x86_64) 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.

vboxuser@Ubuntu:~/RISC$ riscv-none-elf-gdb --version
GNU gdb (xPack GNU RISC-V Embedded GCC x86_64) 15.1
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.

vboxuser@Ubuntu:~/RISC$
```

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

*AI Response:*

### **1. hello.c – Minimal Bare-Metal Code**

```
// hello.c

int main() {

    volatile char *uart = (char *)0x10000000; // Memory-mapped UART address
    const char *msg = "Hello, World!\n";
    while (*msg) {
        *uart = *msg++; // Write each character to UART
    }
    return 0;
}
```

```
// Minimal _start (no C runtime)
```

```
void _start() {
    main();
    while (1); // Trap CPU after main returns
}
```

### **2. linker.ld – Minimal Linker Script**

```
/* linker.ld - Minimal linker script for RISC-V RV32IMC */
```

```
ENTRY(_start)
```

```
SECTIONS {
```

```
    . = 0x80000000; /* Code starts here */
```

```
    . : {
```

```
        *(.*)
    }
```

```
}
```

```
.data : {
```

```
    *(.data*)
```

```
}
```

```
.bss : {
```

```
    *(.bss*)
```

```
    *(COMMON)
```

```
}
```

```
}
```

Save both hello.c and linker.ld in the same directory.

### 3. Compile the ELF Binary

Open terminal in that directory and run:

```
riscv-none-elf-gcc hello.c -o hello.elf \
-march=rv32imc -mabi=ilp32 \
-nostartfiles -nostdlib -T linker.ld
```

### 4. Verify the ELF

```
riscv-none-elf-objdump -d hello.elf
```

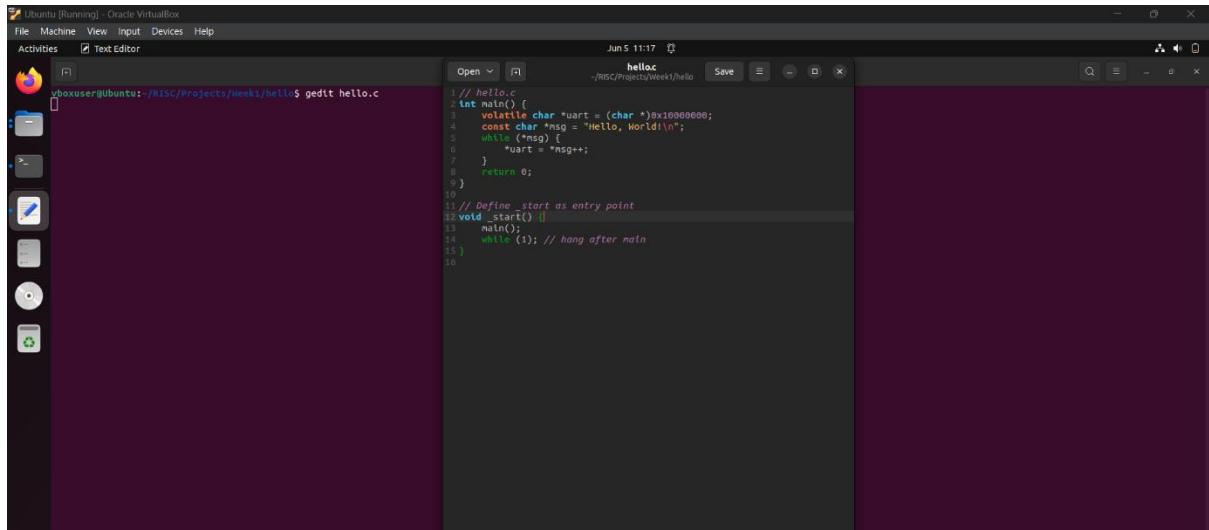
```
riscv-none-elf-readelf -h hello.elf | grep Entry
```

You should see:

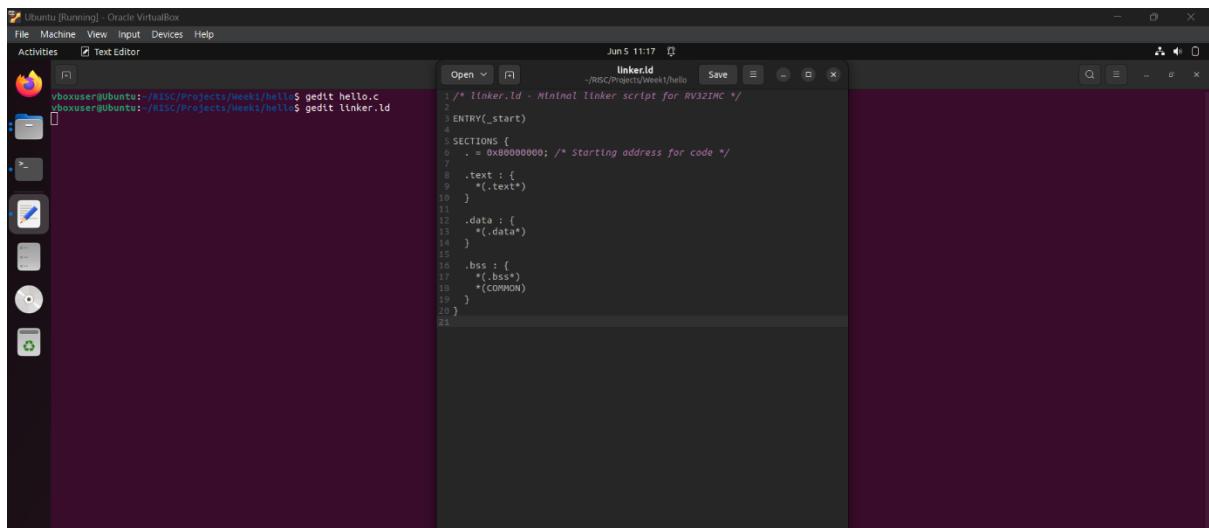
- `_start` in the disassembly
- Entry point set to `_start`, like:

Entry point address: 0x8000004c

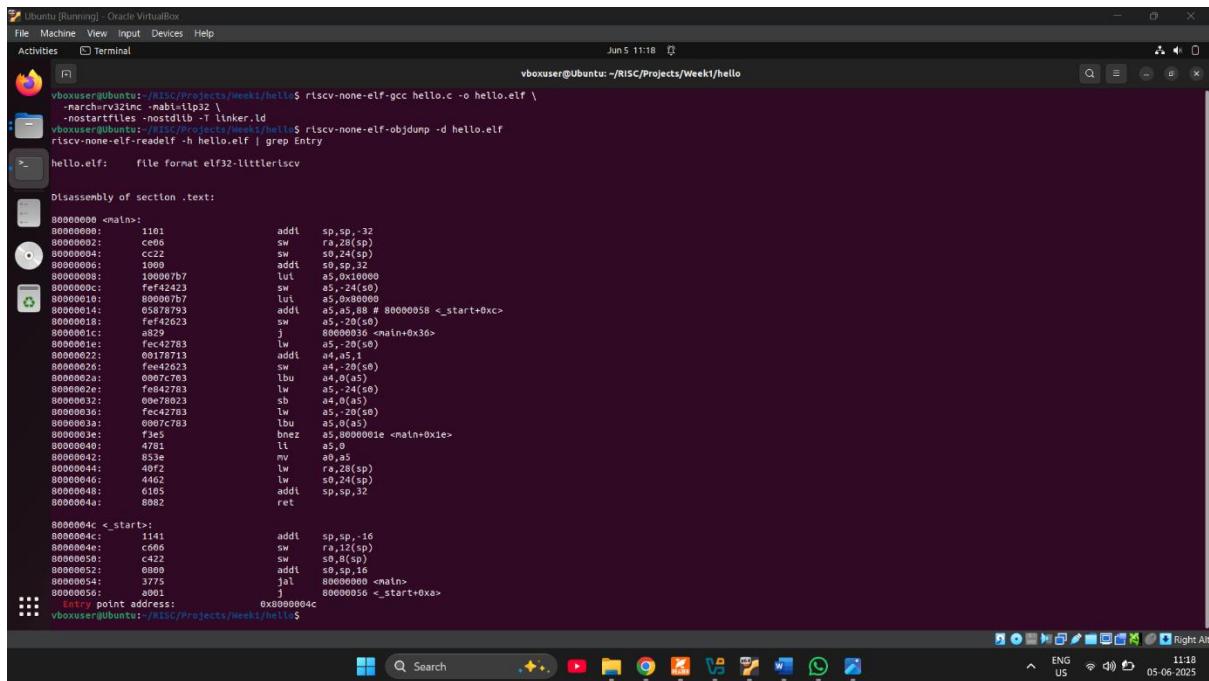
## Output:



```
Ubuntu [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Text Editor Jun 5 11:17
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ gedit hello.c
1 // hello.c
2 int main()
3 {
4     volatile char *uart = (char *)0x10000000;
5     const char *msg = "Hello, World!\n";
6     while (*msg) {
7         *uart = *msg++;
8     }
9 }
10 // Define _start as entry point
11 void _start() {
12     main();
13     while(1); // hang after main
14 }
15 }
```



```
Ubuntu [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Text Editor Jun 5 11:17
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ gedit hello.c
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ gedit linker.ld
1 /* Linker.ld - Minimal Linker script for RV32IMC */
2
3 ENTRY(_start)
4
5 SECTIONS {
6     . = 0x80000000; /* Starting address for code */
7
8     .text : {
9         *(.text*)
10    }
11
12     .data : {
13        *(.data*)
14    }
15
16     .bss : {
17        *(.bss*)
18        *(COMMON)
19    }
20 }
```



```
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ riscv-none-elf-gcc hello.c -o hello.elf \
-march=r32imc -mabi=lp32 \
-nostartfiles -nostdlib -T linker.ld
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ riscv-none-elf-objdump -d hello.elf | grep Entry
hello.elf:   file format elf32-littleriscv

Disassembly of section .text:
00000000 <main>:
00000000: 1101          addi    sp,sp,-32
00000002: ce05          sw      ra,28(sp)
00000004: c222          sw      s0,0(sp)
00000005: 1000            addi    s0,sp,32
00000009: 100007b7        lut    a5,0x10000
0000000c: fe42423        sw      a5,24(s0)
00000010: 800007b7        lut    a5,0x80000
00000011: 00000793        addi    a5,35,0x10000058 <_start+0xc>
00000013: fa42623        sw      a5,24(s0)
00000015: a829           j      80000036 <main+0x36>
00000016: fec42783        lw      a5,20(s0)
00000017: 09178713        addt   a4,a5,16
00000019: fe42623        sw      a4,24(s0)
00000021: 09178723        lbu   a4,(s0)
00000022: fe842783        lw      a5,24(s0)
00000023: 00e78823        sb      a5,0(a5)
00000024: fec42783        lw      a5,20(s0)
00000025: 0000c783        lbu   a5,0(a5)
00000026: f3e5           bnez  a5,0x0000001e <main+0x1e>
00000040: 4781           li      a3,9
00000042: 853e           mv      a9,a5
00000044: 40f2           lw      ra,28(sp)
00000046: 4462           lw      s0,24(sp)
00000048: 6105           addi   s0,sp,32
0000004a: 8082           ret

0000004c <_start>:
0000004c: 1101          addi    sp,sp,-16
0000004e: c006          sw      ra,28(sp)
00000050: c422          sw      s0,0(sp)
00000052: 0000            addt   s0,sp,16
00000054: 3775           jal    80000000 <main>
00000056: a001           j      80000056 <_start+0xa>
entry point address: 0x8000004c
```

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

**AI Response:**

### **Step 1: Generate the .s File**

You can use the following command:

```
riscv-none-elf-gcc hello.c -S -o hello.s \
-march=rv32imc -mabi=ilp32 \
-nostartfiles -nostdlib -T linker.ld
```

- **-S:** Tells GCC to **generate assembly** (.s) instead of object or ELF.
- **-o hello.s:** Output file
- Other flags are same as before (bare-metal, RV32IMC).

This will produce a file called hello.s.

### **Step 2: Look at the Assembly (hello.s)**

Here's an example snippet you'll likely see for the main() function (or similar, depending on GCC version):

main:

```
addi sp,sp,-32      # Allocate 32 bytes stack
sw   ra,28(sp)      # Save return address
sw   s0,24(sp)      # Save frame pointer
addi s0,sp,32       # Set up frame pointer (s0 = old sp)

...
lw   ra,28(sp)      # Restore return address
lw   s0,24(sp)      # Restore frame pointer
addi sp,sp,32       # Deallocate stack
ret                # Return
```

### **Step 3: Explain Prologue and Epilogue**

#### **◆ Prologue (Function Entry)**

```
addi sp,sp,-32      # Allocate stack space
sw   ra,28(sp)      # Save return address
sw   s0,24(sp)      # Save frame pointer (s0)
```

```
addi s0,sp,32      # Set new frame pointer
```

This does:

- Allocates 32 bytes on the stack
- Saves ra (return address) and s0 (caller's frame pointer)
- Sets s0 to the old stack pointer + 32 (new frame pointer)

🔧 This is **standard calling convention** for RISC-V (ilp32) so that:

- The function can safely call other functions
- Local variables and call frames are isolated

#### ◆ **Epilogue (Function Exit)**

```
lw    ra,28(sp)    # Restore return address  
lw    s0,24(sp)    # Restore old frame pointer  
addi sp,sp,32      # Free stack space  
ret             # Return to caller
```

This does:

- Restores ra and s0
- Frees up the stack space used in prologue
- Returns control to the caller using ret

## Output:

The screenshot shows a Linux desktop environment with a terminal window and a code editor.

In the terminal window (top), the user is in a directory under "/RISC/Projects/Week1/hello". They run "ls" to list files, then "riscv-none-elf-gcc hello.c -S" to generate assembly code. The assembly output is then piped to "gedit" for viewing.

```
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ ls
hello.c  hello.elf  linker.ld
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ riscv-none-elf-gcc hello.c -S > hello.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ gedit hello.s
```

The code editor window (bottom) displays the generated assembly code for the "hello" program. The assembly code starts with the .file directive and includes sections for .text, .data, and .bss, along with various instructions like addi, sw, lw, and bne.

```
.file "hello.c"
.option nounicode
.attributes arch, "rv32ilp32,m2p0,c2p0"
.attributes unaligned_access, 0
.attribute stack_align, 16
.text
.section .rodata
.align 2
.LC0:
.string "Hello, World!\n"
.text
.align 1
.global main
.type main, @function
main:
.addi sp,sp,-32
.sw r0,28($sp)
.sw $0,24($sp)
.addi $0,$p,-32
.ll a5,268435456
.sw a5,24($sp)
.lui a5,4000(LC0)
.addi a5,a5,10($LC0)
.sw a5,20($sp)
.j .L2
.L1:
.lw a5,-20($sp)
.addi a5,55,-20
.sw a5,24($sp)
.lbu a4,0($a5)
.lw a5,24($sp)
.sb a4,0($a5)
.L2:
.lw a5,-20($sp)
.lbu a5,0($a5)
.bne a5,zero,.L3
```

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

*AI Response:*

## **1. Convert ELF to Raw Binary or Hex**

### **Option A: Generate .bin (raw binary)**

```
riscv-none-elf-obj -O binary hello.elf hello.bin
```

This produces a flat binary with no headers, suitable for loading into memory directly.

### **Option B: Generate .hex (Intel HEX format)**

```
riscv-none-elf-obj -O ihex hello.elf hello.hex
```

This produces an Intel HEX file, often used by flash programmers.

## **2. Disassemble the ELF File**

Use objdump to disassemble the ELF:

```
riscv-none-elf-objdump -d hello.elf
```

To include source code (if available), use:

```
riscv-none-elf-objdump -S hello.elf
```

## **3. Understanding the Disassembly Output**

Example output:

```
80000000 <main>:
```

```
80000000: 1101      addi  sp,sp,-32
```

```
80000002: ce06      sw    ra,28(sp)
```

```
80000004: cc22      sw    s0,24(sp)
```

Each line has:

<b>Column</b>	<b>Description</b>
80000000:	Address of the instruction in memory
1101	Machine code in hexadecimal
addi sp,sp,-32	Assembly instruction in human-readable form

This shows the exact encoding of each instruction and its location in memory.

#### **4. View Binary Contents in Hex**

You can inspect the raw binary contents:

```
hexdump -C hello.bin
```

This outputs the binary data in hexadecimal and ASCII, for example:

00000000 13 01 01 f0 23 2e 11 00 23 2c 81 00 13 81 01 20

Each pair of hex digits corresponds to 1 byte, and you can match them against the instruction opcodes.

## Output:

*List all 32 RV32 integer registers with their ABI names and typical calling-convention roles.*

*AI Response:*

Here's a complete list of the **32 integer registers in RV32I** with:

- **Register number**
- **ABI name**
- **Conventional role in calling convention**

### **RV32I Integer Register Table**

#### **Reg No ABI Name Role / Usage**

x0	zero	Constant zero (hardwired, always 0)
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary register 0
x6	t1	Temporary register 1
x7	t2	Temporary register 2
x8	s0 / fp	Saved register 0 / Frame pointer
x9	s1	Saved register 1
x10	a0	Argument 0 / Return value
x11	a1	Argument 1 / Return value
x12	a2	Argument 2
x13	a3	Argument 3
x14	a4	Argument 4
x15	a5	Argument 5
x16	a6	Argument 6
x17	a7	Argument 7

### **Reg No ABI Name Role / Usage**

x18	s2	Saved register 2
x19	s3	Saved register 3
x20	s4	Saved register 4
x21	s5	Saved register 5
x22	s6	Saved register 6
x23	s7	Saved register 7
x24	s8	Saved register 8
x25	s9	Saved register 9
x26	s10	Saved register 10
x27	s11	Saved register 11
x28	t3	Temporary register 3
x29	t4	Temporary register 4
x30	t5	Temporary register 5
x31	t6	Temporary register 6

### **Calling Convention Summary**

- **Caller-saved:** a0–a7, t0–t6

These may be overwritten by a called function. If a caller wants to preserve them, it must save them.

- **Callee-saved:** s0–s11

These must be preserved by the called function across calls.

- **Special-purpose:**

- x0 (zero): Always returns 0
- x1 (ra): Stores return address from a function call
- x2 (sp): Points to top of the stack
- x8 (s0/fp): Often used as frame pointer for stack frames

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

**AI response:**

**Compile:**

```
riscv-none-elf-gcc -march=rv32im -mabi=ilp32 -nostdlib -T linker.ld -o hello2.elf hello2.c
```

**Verify ELF:**

```
riscv-none-elf-readelf -h hello2.elf
```

**Debug with GDB**

**Start GDB:**

```
riscv-none-elf-gdb hello2.elf
```

```
(gdb) set breakpoint auto-hw off
```

```
(gdb) target sim
```

Connected to the simulator.

```
(gdb) load
```

```
Loading section ., size 0x... lma 0x80000000
```

```
Start address 0x80000000
```

```
(gdb) info breakpoints
```

```
(gdb) delete
```

```
(gdb) break main
```

```
Breakpoint 1 at 0x80000000: file hello2.c, line 2.
```

```
(gdb) run
```

```
Starting program: /home/vboxuser/RISC/Projects/Week1/hello/hello2.elf
```

```
Breakpoint 1, main () at hello2.c:2
```

```
2      volatile int x = 42;
```

```
(gdb) step
```

```
3      x = x + 1;
```

```
(gdb) stepi
```

```
(gdb) info registers
```

```
(gdb) step
```

```
4      return x;
```

```
(gdb) print x
```

```
$4 = 43
```

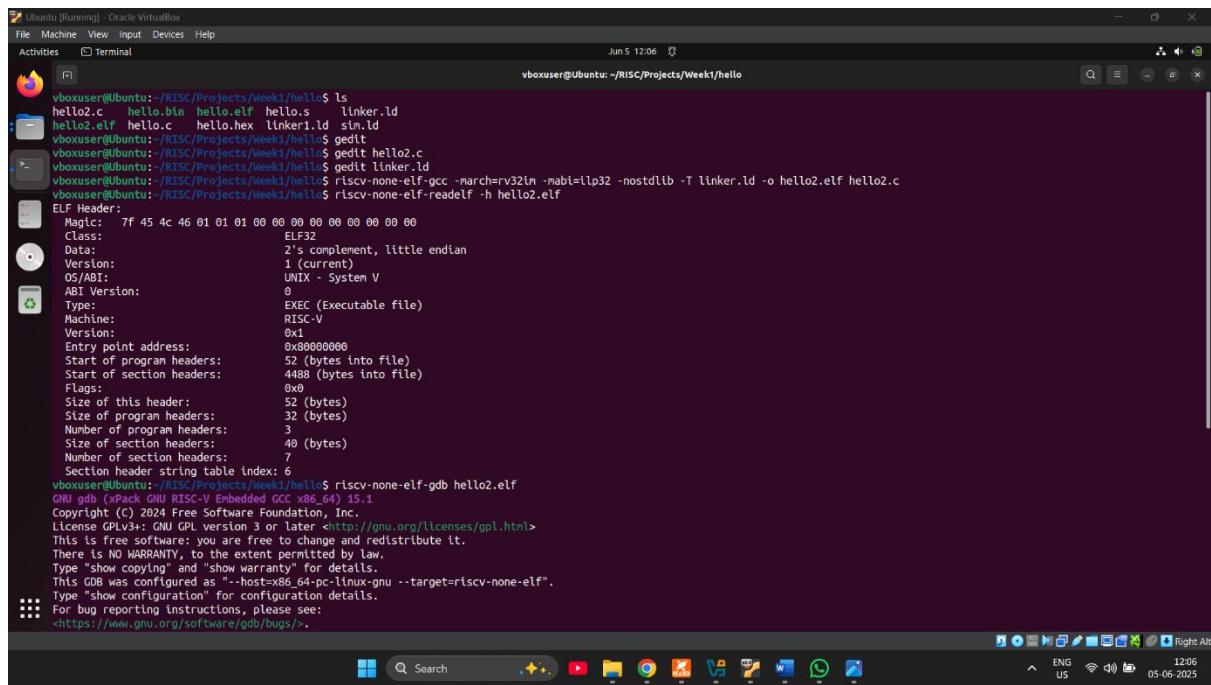
```
(gdb) print $a0
```

```
$5 = 43
```

```
(gdb) continue
```

```
(gdb) quit
```

## Output:



The screenshot shows a terminal window titled "vboxuser@Ubuntu: ~/RISC/Projects/Week1/hello". The terminal displays the following command-line session:

```
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ ls
hello2.c  hello.bin  hello.elf  hello.s  linker.ld
hello2.elf  hello.c  hello.hex  linker1.ld  sim.ld
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ gedit hello2.c
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ gedit linker.ld
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ riscv-none-elf-gcc -march=rv32im -mabi=lp32 -nostdlib -T linker.ld -o hello2.elf hello2.c
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ riscv-none-elf-readelf -h hello2.elf
ELF Header:
  Magic: 7f 45 4c 46 01 01 00 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: 4488 (bytes into file)
  Flags: 0x8
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 3
  Size of section headers: 40 (bytes)
  Number of section headers: 7
  Section header string table index: 6
vboxuser@Ubuntu:~/RISC/Projects/Week1/hello$ riscv-none-elf-gdb hello2.elf
GNU gdb (XPack GNU RISC-V Embedded GCC x86_64) 15.1
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 for "-host=x86_64-pc-linux-gnu --target=riscv-none-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
```

```
Ubuntu [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Terminal
vboxuser@Ubuntu: ~/RISC/Projects/Week1/hello Jun 5 12:07
vboxuser@Ubuntu: ~/RISC/Projects/Week1/hello

Size of program headers:      32 (bytes)
Number of program headers:    3
Size of section headers:     40 (bytes)
Number of section headers:   7
Section header string table index: 6
vboxuser@Ubuntu: ~/RISC/Projects/Week1/hello$ riscv-none-elf-gdb hello2.elf
GNU gdb (xPack GNU RISC-V Embedded GCC x86_64) 15.1
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=riscv-none-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 hello2.elf...
(No debugging symbols found in hello2.elf)
(gdb) set breakpoint auto-hw off
(gdb) target sim
Connected to the simulator.
(gdb) load
Loading section .text, size 0x3c lma 80000000
Start address 80000000
Transfer rate: 480 bits in <1 sec.
(gdb) info breakpoints
No breakpoints, watchpoints, tracepoints, or catchpoints.
(gdb) delete
(gdb) break main
Breakpoint 1 at 0x80000000
(gdb) info breakpoints
Num Type Disp Enb Address What
::: 1 breakpoint keep y 0x80000000 <main>
(gdb) step

::: Num Type Disp Enb Address What
::: 1 breakpoint keep y 0x80000000 <main>


```

```
Ubuntu [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Activities Terminal
vboxuser@Ubuntu: ~/RISC/Projects/Week1/hello Jun 5 12:09
vboxuser@Ubuntu: ~/RISC/Projects/Week1/hello

0x80000000 in main ()
(gdb) info registers
r0      0x0      0x0
sp      0xfffff150  0x3fff150
gp      0x0      0x0
t0      0x0      0
t1      0x0      0
t2      0x0      0
fp      0x0      0x0
s1      0x0      0
a0      0x1      1
t1      0x0      0
s2      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
s8      0x0      0
s9      0x0      0
s10     0x0      0
s11     0x0      0
t3      0x0      0
t4      0x0      0
t5      0x0      0
t6      0x0      0
pc      0x80000000  0x80000000 <main>
(gdb) print $x10
$1 = 1
(gdb) print $pc
$2 = (void (*)()) 0x80000000 <main>
(gdb)


```

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

*AI response:*

**1. Compile:**

```
riscv-none-elf-gcc -g -march=rv32im -mabi=ilp32 -nostdlib -T linker.ld -o hello2.elf hello2.c  
startup.s
```

- The updated linker.ld should eliminate the “non-existent phdr” errors.
- If errors persist, double-check linker.ld syntax.

**2. Verify ELF:**

```
riscv-none-elf-readelf -h hello2.elf
```

Confirm:

- Machine: RISC-V
- Class: ELF32
- Entry point address: 0x80000000

Check program headers and sections:

```
riscv-none-elf-readelf -l hello2.elf
```

Ensure:

- . and .rodata in FLASH with R E flags.
- .data and .bss in RAM with R W flags.

## **Run with QEMU**

Run the ELF in QEMU’s virt machine:

```
qemu-system-riscv32 -nographic -machine virt -bios none -kernel  
~/RISC/Projects/Week1/uart/hello2.elf
```

- Expected output:

Printing from UART

Value of x: 43

- Options:

- -nographic: Sends UART output to the terminal.
- -machine virt: Uses UART at 0x10000000.
- -bios none: Bare-metal, no BIOS.

- -kernel: Loads the ELF.

## Output:

The screenshot shows a Linux desktop environment with a dark theme. On the left is a file manager window showing files like hello2.c, hello2.elf, linker.ld, start.S, and startup.s. In the center, a terminal window is open with the command `vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit hello2.c`. To the right is a code editor window titled "hello2.c" showing C code for a UART driver. The code defines constants for UART\_TX and UART\_READY, and implements functions for putting characters and strings over the UART. It includes a main loop that prints the value of x to the UART.

```

1 #define UART_TX 0x10000000 // UART transmit register for OEMU virt machine
2 #define UART_READY 0x10000005 // UART status register (bit 5 = TX ready)
3
4 void uart_putchar(char c) {
5     volatile char *uart_tx = (volatile char*)UART_TX;
6     volatile char *uart_ready = (volatile char*)UART_READY;
7     while (!(*uart_ready & (1 << 5))); // Wait for TXIFO empty (bit 5)
8     *uart_tx = c; // Write character to UART
9 }
10
11 void uart_puts(const char* s) {
12     while (*s) {
13         uart_putchar(*s++);
14     }
15 }
16
17 int main() {
18     volatile int x = 42;
19     x = x + 1;
20     uart_puts("Printing from UART");
21     uart_puts("Value of x: ");
22     // Convert x to string (simple decimal conversion)
23     if (x == 0) {
24         uart_putchar('0');
25     } else {
26         char buf[10];
27         int i = 0;
28         while (x > 0) {
29             buf[i++] = '0' + (x % 10);
30             x /= 10;
31         }
32         while (i > 0) {
33             uart_putchar(buf[--i]);
34         }
35     }
36     uart_puts("\n");
37     return x;
38 }

```

The screenshot shows a Linux desktop environment with a dark theme. On the left is a file manager window showing files like hello2.c, hello2.elf, linker.ld, start.S, and startup.s. In the center, a terminal window is open with the command `vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit hello2.c`. To the right is a code editor window titled "startup.s" showing assembly code for the startup script. The assembly code initializes the stack pointer, calls the main function, and enters an infinite loop.

```

1 .section .text.start
2 .global _start
3 _start:
4     la sp, _stack_top    # Initialize stack pointer
5     jal main             # Call main
6     b .                  # Infinite loop after main returns
7 .section .bss
8 .align 4
9 .space 1024            # Reserve 1KB for stack
10 _stack_top:

```

```
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit hello2.c

vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit start.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ riscv-none-elf-gcc -g -march=r32im -mabi=lp32 -nostdlib -T linker.ld -o hello2.elf hello2.c startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ riscv-none-elf-readelf -l 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: 6824 (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: 16
Section header string table index: 15
Section header string table index: 15
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ riscv-none-elf-readelf -l hello2.elf

Elf file type is EXEC (Executable file)
Entry point 0x80000000
There are 4 program headers, starting at offset 52

Program Headers:
  Type           Offset     VirtAddr PhysAddr  FileSiz MemSiz Flg Align
  RISCV_ATTRIBUTES    0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 R E 0x1000
  LOAD          0x001000 0x80000000 0x80000000 0x000210 0x000210 R E 0x1000
  LOAD          0x000000 0x81000000 0x800000218 0x000000 0x000400 RW 0x1000
  GNU_STACK    0x000000 0x00000000 0x00000000 0x000000 0x000000 RW 0x1000

Section to Segment mapping:
  Segment Sections...
  00  .riscv.attributes
  01  .text .rodata
  02  .bss
  03

vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$
```

```
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit start.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ riscv-none-elf-gcc -g -march=r32im -mabi=lp32 -nostdlib -T linker.ld -o hello2.elf hello2.c startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ riscv-none-elf-readelf -l 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: 6824 (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: 16
Section header string table index: 15
Section header string table index: 15
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ riscv-none-elf-readelf -l hello2.elf

Elf file type is EXEC (Executable file)
Entry point 0x80000000
There are 4 program headers, starting at offset 52

Program Headers:
  Type           Offset     VirtAddr PhysAddr  FileSiz MemSiz Flg Align
  RISCV_ATTRIBUTES    0x001782 0x00000000 0x00000000 0x0002a 0x0002a R 0x1
  LOAD          0x001000 0x80000000 0x80000000 0x000210 0x000210 R E 0x1000
  LOAD          0x000000 0x81000000 0x800000218 0x000000 0x000400 RW 0x1000
  GNU_STACK    0x000000 0x00000000 0x00000000 0x000000 0x000000 RW 0x1000

Section to Segment mapping:
  Segment Sections...
  00  .riscv.attributes
  01  .text .rodata
  02  .bss
  03

vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel ~/RISC/Projects/Week1/uart/hello2.elf
Printing from UARTValue of x: 43
```

```
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel ~/RISC/Projects/Week1/uart/hello2.elf
Printing from UARTValue of x: 43
```

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

*AI response:*

**1. Navigate to Directory:**

```
cd ~/RISC/Projects/Week1/uart
```

**2. Compile with -O0 (No Optimization):**

- Generate the ELF:

```
riscv-none-elf-gcc -g -O0 -march=rv32im -mabi=ilp32 -nostdlib -T linker.ld -o hello2_O0.elf  
hello2.c startup.s
```

- Generate assembly:

```
riscv-none-elf-gcc -g -O0 -march=rv32im -mabi=ilp32 -nostdlib -S -o hello2_O0.s hello2.c
```

The -S flag produces assembly output (hello2\_O0.s) for hello2.c. startup.s is already in assembly.

**3. Compile with -O2 (Optimization Level 2):**

- Generate the ELF:

```
riscv-none-elf-gcc -g -O2 -march=rv32im -mabi=ilp32 -nostdlib -T linker.ld -o hello2_O2.elf  
hello2.c startup.s
```

- Generate assembly:

```
riscv-none-elf-gcc -g -O2 -march=rv32im -mabi=ilp32 -nostdlib -S -o hello2_O2.s hello2.c
```

**4. Verify Compilation:** Ensure no errors (e.g., linker issues). The provided linker.ld should prevent the previous “non-existent phdr” errors. Check ELFs:

```
riscv-none-elf-readelf -h hello2_O0.elf
```

```
riscv-none-elf-readelf -h hello2_O2.elf
```

**Run the ELFs to Confirm Functionality**

Test both ELFs in QEMU to ensure they produce the expected UART output:

```
qemu-system-riscv32 -nographic -machine virt -bios none -kernel  
~/RISC/Projects/Week1/uart/hello2_O0.elf
```

```
qemu-system-riscv32 -nographic -machine virt -bios none -kernel  
~/RISC/Projects/Week1/uart/hello2_O2.elf
```

- Expected output for both:

Value of x: 43

## Differences and Why They Occur

### 1. Stack Frame Size:

- **-O0:** Allocates a 48-byte stack frame to store x, i, buf, and saved registers (ra, s0), even if some aren't needed.
- **-O2:** Uses a 16-byte stack frame, only allocating space for buf when needed and saving fewer registers.
- **Why:** -O0 prioritizes debuggability, storing all variables on the stack. -O2 minimizes stack usage by keeping variables in registers (t1, t3) and allocating buf dynamically.

### 2. Variable Storage:

- **-O0:** Stores x and i on the stack (-20(s0), -24(s0)), with frequent loads (lw) and stores (sw).
- **-O2:** Keeps x in t1 and i in t2, avoiding stack access except for buf.
- **Why:** -O0 avoids register optimization to ensure variables are accessible for debugging. -O2 uses registers to reduce memory access latency.

### 3. Instruction Count:

- **-O0:** More instructions due to redundant loads/stores and verbose stack management (e.g., sw s0, lw s0).
- **-O2:** Fewer instructions by eliminating redundant operations and optimizing control flow.
- **Why:** -O0 translates C code literally, while -O2 applies optimizations like register allocation, dead code elimination, and loop tightening.

### 4. Loop Efficiency:

- **-O0:** The number conversion loop (while (x > 0) and while (i > 0)) uses stack-based addressing for buf and repeated loads for x and i.
- **-O2:** Uses a single stack slot for buf and keeps x and i in registers, reducing memory operations.
- **Why:** -O2 optimizes loops by minimizing memory accesses and reusing registers.

### 5. Function Calls:

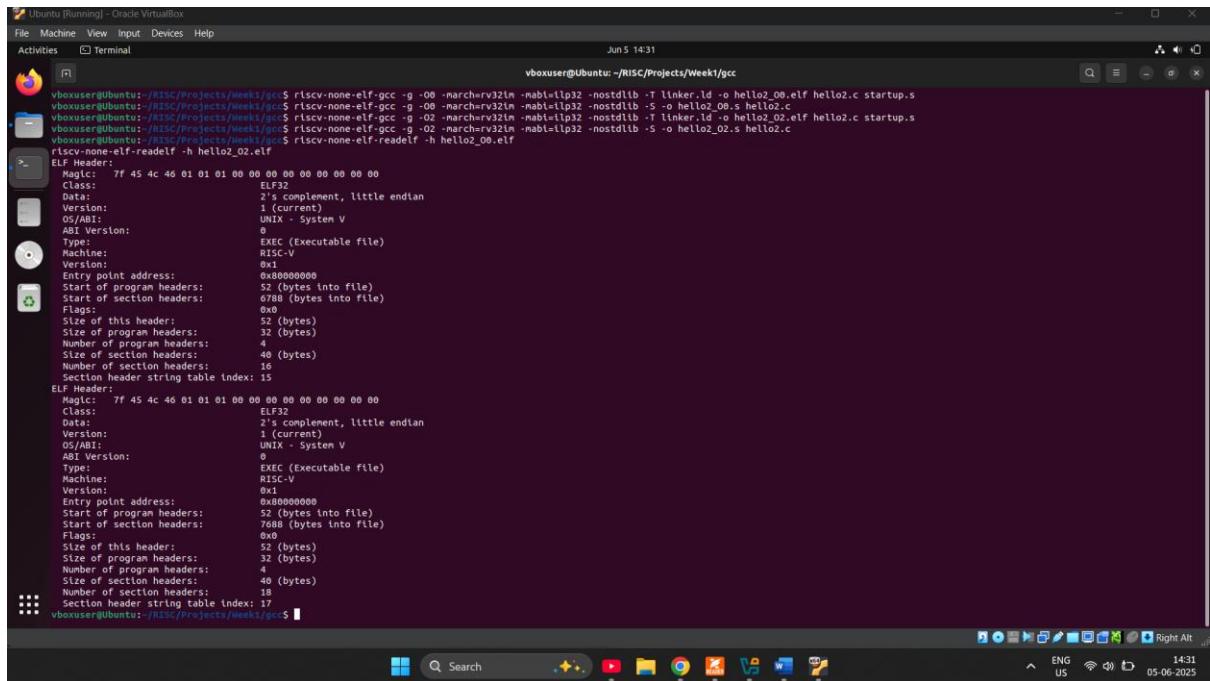
- **-O0:** Explicit calls to uart\_putc and uart\_puts with full stack frame setup.
- **-O2:** Similar calls, but the surrounding code is optimized (e.g., fewer stack operations).

- **Why:** -O0 avoids inlining to preserve function boundaries for debugging. -O2 could inline small functions, but here, uart\_putchar and uart\_puts remain separate due to their complexity.

## 6. Debug Information:

- **-O0:** Includes detailed debug info (e.g., line numbers, variable locations), making it easier to trace in GDB.
- **-O2:** Retains some debug info (since -g is used), but optimized code may skip some variable updates, complicating debugging.
- **Why:** -O0 is designed for debugging, preserving all C-level constructs. -O2 prioritizes performance, potentially reordering or eliminating operations.

## Output:



```
vboxuser@Ubuntu:~/RISC/Projects/Week1/gcc$ riscv-none-elf-gcc -g -O0 -march=r32f -mabi=lp32 -nostdlib -T linker.ld -o hello2_00.elf hello2.c startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/gcc$ riscv-none-elf-gcc -g -O0 -march=r32f -mabi=lp32 -nostdlib -S -o hello2_00.s hello2.c
vboxuser@Ubuntu:~/RISC/Projects/Week1/gcc$ riscv-none-elf-gcc -g -O2 -march=r32f -mabi=lp32 -nostdlib -T linker.ld -o hello2_02.elf hello2.c startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/gcc$ riscv-none-elf-readelf -h hello2_00.elf
ELF Header:
  Magic: 7f 45 4c 46 01 01 00 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: 0x788 (bytes into file)
  Start of section headers: 0x788 (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: 16
  Section header string table index: 17
ELF Header:
  Magic: 7f 45 4c 46 01 01 00 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: 0x788 (bytes into file)
  Start of section headers: 0x788 (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: 18
  Section header string table index: 17
vboxuser@Ubuntu:~/RISC/Projects/Week1/gcc$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel ~/RISC/Projects/Week1/gcc/hello2_02.elf
Value of x: 43

vboxuser@Ubuntu:~/RISC/Projects/Week1/gcc$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel ~/RISC/Projects/Week1/gcc/hello2_02.elf
Value of x: 43
```

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

*AI response:*

## Compilation and Testing

### 1. Save Files:

- Save the updated hello2.c, startup.s, and linker.ld in ~/RISC/Projects/Week1/uart/.
- Ensure you're in the correct directory:

```
cd ~/RISC/Projects/Week1/uart
```

### 2. Compile:

```
riscv-none-elf-gcc -g -O0 -march=rv32im -mabi=ilp32 -nostdlib -T linker.ld -o hello2.elf  
hello2.c startup.s
```

### 3. Run in QEMU:

```
qemu-system-riscv32 -nographic -machine virt -bios none -kernel  
~/RISC/Projects/Week1/uart/hello2.elf
```

- Expected output (cycle count varies):

## Inline Assembly Breakdown

The inline assembly statement is:

```
__asm__ volatile (  
    "rdcycle %0"  
    : "=r" (cycles)  
    :  
    :  
);
```

Let's break down each part, focusing on the constraints:

### 1. Assembly Template:

- "rdcycle %0": The actual assembly instruction. rdcycle reads the CYCLE CSR (0xC00) into a register. %0 is a placeholder for the output operand (cycles).

### 2. Output Constraint: : "=<r>" (<cycles>)

- **Syntax:** =r (output\_operand)
- **Explanation:**

- = : Indicates the operand is write-only (the instruction writes to it).
- r : Specifies a general-purpose register (e.g., x0-x31 in RISC-V). The compiler allocates a register to store the result of rdcycle.
- (cycles) : Binds the result to the C variable cycles (a uint32\_t). After rdcycle executes, the value in the allocated register is copied to cycles.
- **Purpose:** Ensures the cycle count is stored in the cycles variable for return.

### 3. Input Constraints: : (empty)

- **Explanation:** No input operands are needed because rdcycle doesn't require input values—it directly reads the CSR.
- **Purpose:** The empty input constraint list indicates the instruction operates independently of C-level inputs.

### 4. Clobbered Registers: : (empty)

- **Explanation:** No additional registers are modified (clobbered) by rdcycle beyond the output register (%0). The instruction only writes to the destination register and doesn't affect other registers or memory.
- **Purpose:** Informs the compiler that no registers need to be saved/restored, simplifying code generation.

### 5. \_\_asm\_\_ volatile:

- **\_\_asm\_\_:** GCC keyword for inline assembly.
- **volatile:** Prevents the compiler from optimizing away the assembly (e.g., removing it if it thinks the output is unused).
- **Purpose:** Ensures the rdcycle instruction is executed exactly as written.

### Expected Output:

Value of x: 43

Cycles taken: <some number>

## Output:

```
#include <stdint.h>
...
4 uint32_t read_cycle_counter(void) {
5     uint32_t cycles;
6     __asm__ volatile (
7         "rdcycle %0"
8         : "=r" (cycles) // Output constraint
9         : // No input constraints
10        : // No clobbered registers
11    );
12    return cycles;
13 }
```

```
#include <stdint.h>
...
4 #define UART_TX 0x10000000 // UART transmit register for QEMU virt machine
4 #define UART_READY 0x10000005 // UART status register (bit 5 = TX ready)
5
6 void uart_putc(char c) {
7     volatile char *uart_tx = (volatile char*)UART_TX;
8     volatile char *uart_ready = (volatile char*)UART_READY;
9     while (!(*uart_ready & (1 << 5))); // Wait for TXFIFO empty (bit 5)
10    *uart_tx = c;
11 }
12
13 void uart_puts(const char* s) {
14     while (*s) {
15         uart_putc(*s++);
16     }
17 }
18
19 uint32_t read_cycle_counter(void) {
20     uint32_t cycles;
21     __asm__ volatile (
22         "rdcycle %0"
23         : "=r" (cycles)
24         : ;
25    );
26    return cycles;
27 }
28
29 void uart_put_num(uint32_t num) {
30     if (num == 0) {
31         uart_putc('0');
32     } else {
33         char buf[10];
34         int i = 0;
35         while (num > 0) {
36             buf[i++] = '0' + (num % 10);
37             num /= 10;
38         }
39         while (i > 0) {
40             uart_putc(buf[--i]);
41         }
42     }
43 }
44
45 int main() {
```

```
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit cycle_counter.c
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ gedit hello2.c
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ riscv-none-elf-gcc -g -O0 -march=rv32im -mabi=lp32 -nostdlib -T linker.ld -o hello2.elf hello2.c startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/uart$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel ~/RISC/Projects/Week1/uart/hello2.elf
Value of x: 43
Cycles taken: 1374688
```

*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?*

**AI Response:**

## Compilation and Testing

### 1. Save Files:

- Save gpio\_toggle.c, startup.s, and linker.ld in ~/RISC/Projects/Week1/uart/.
- Navigate to:

```
cd ~/RISC/Projects/Week1/uart
```

### 2. Compile:

```
riscv-none-elf-gcc -g -O2 -march=rv32im -mabi=ilp32 -nostdlib -T linker.ld -o gpio_toggle.elf gpio_toggle.c startup.s
```

### 3. Verify ELF:

```
riscv-none-elf-readelf -h gpio_toggle.elf
```

Confirm Machine: RISC-V, Class: ELF32, Entry point address: 0x80000000.

### 4. Run in QEMU:

```
qemu-system-riscv32 -nographic -machine virt -bios none -kernel  
~/RISC/Projects/Week1/uart/gpio_toggle.elf
```

#### ○ Expected Output:

GPIO Toggled

..... (dots continue)

- **Previous Issue:** Only "AG" printed due to a fault at 0x10012000. If "A GPIO Toggled B ...." prints, the write-only approach and early print avoid the fault.
- **If Issue Persists:** If only "A GPIO Toggled" prints, a write to 0x10012000 is faulting.

### 5. Test with Spike:

```
spike --isa=rv32im ~/RISC/Projects/Week1/uart/gpio_toggle.elf
```

- Expected: A, GPIO Toggled, B, dots.
- Spike may ignore writes to 0x10012000.

## Output:

The screenshot shows a terminal window titled "Ubuntu [Running] - Oracle VM VirtualBox" with the command line interface. The terminal displays the following output:

```
vboxuser@ubuntu:/RISC/Projects/Week1/uart$ riscv-none-elf-gcc -g -O2 -march=rv32im -mabi=lp32 -nostdlib -T linker.ld -o gpio_toggle.elf gpio_toggle.c startup.s
vboxuser@ubuntu:/RISC/Projects/Week1/uart$ riscv-none-elf-readelf -h gpio_toggle.elf
ELF Header:
  Magic: 7f 4c 46 01 01 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: 0x4000000000000000
Start of program headers: 52 (bytes into file)
Start of section headers: 7676 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 1
Size of section headers: 40 (bytes)
Number of section headers: 18
Section header string table index: 17
vboxuser@ubuntu:/RISC/Projects/Week1/uart$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel ./RISC/Projects/Week1/uart/gpio_toggle.elf
GPIO Toggled
```

The terminal window is part of a desktop environment, with a taskbar at the bottom showing various application icons like a search bar, file manager, browser, and terminal.

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

*AI Response:*

- . at address 0x00000000
- .data at address 0x10000000

## Compile Command

- riscv-none-elf-gcc -march=rv32imc -mabi=ilp32 \
- -nostdlib -nostartfiles -T linker.ld \
- \_start.s hello.c -o hello.elf
- Make sure \_start.s contains the \_start label and sets up the stack and jumps to main.

The screenshot shows a Linux desktop environment with a terminal window and a text editor window. The terminal window displays the following command and its output:

```
vboxuser@Ubuntu: ~/RISC/Projects/Week1/link$ riscv-none-elf-gcc -g -march=rv32imc -mabi=ilp32 -nostdlib -T linker.ld -o hello2.elf hello2.c startup.s
vboxuser@Ubuntu: ~/RISC/Projects/Week1/link$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel ./hello2.elf
Printing from UARTValue of x: 43
```

The text editor window shows a linker script named `linker.ld` with the following content:

```
OUTPUT_ARCH(riscv)
ENTRY(_start)
MEMORY
{
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 256K
    RAM (rw) : ORIGIN = 0x10000000, LENGTH = 64K
}
SECTIONS
{
    . = ORIGIN(FLASH);
    .text : (
        .text.start
        *(.text)
        *(.text.*)
    ) > FLASH
    .rodata : ALIGN(4) {
        *(.rodata)
        *(.rodata.*)
    } > FLASH
    .data : ALIGN(4) {
        *(.data)
        *(.data.*)
    } > RAM AT > FLASH
    .bss : ALIGN(4) {
        *(.bss)
        *(.bss.*)
    } > RAM
    _end = .;
```

The screenshot shows a Linux terminal window with the following command and its output:

```
vboxuser@Ubuntu: ~/RISC/Projects/Week1/link$ riscv-none-elf-gcc -g -march=rv32imc -mabi=ilp32 -nostdlib -T linker.ld -o hello2.elf hello2.c startup.s
vboxuser@Ubuntu: ~/RISC/Projects/Week1/link$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel ./hello2.elf
Printing from UARTValue of x: 43
```

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

*AI Response:*

In a bare-metal RISC-V program, crt0.S (also called C runtime zero) is the first assembly file that runs after reset. It sets up the execution environment before calling your main() function. It is essential when you're not using a full C library like newlib or glibc.

---



**What crt0.S typically does:**

#### **Step Task**

- 1 Set up the stack pointer (sp)**
- 2 Initialize .bss (zero out uninitialized data)**
- 3 Optionally .data from flash to RAM**
- 4 Call the main() function**
- 5 Handle main() return (often enters infinite loop or triggers exit routine)**

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

**AI response:**

## Compilation and Testing

### 1. Save Files:

- Save timer\_interrupt.c, trap\_handler.s, startup.s, linker.ld in ~/RISC/Projects/Week1/MTIP/.
- Navigate:

```
cd ~/RISC/Projects/Week1/MTIP
```

### 2. Compile:

```
riscv-none-elf-gcc -g -O2 -march=rv32imc_zicsr -mabi=ilp32 -nostdlib -T linker.ld -o timer.elf timer_interrupt.c trap_handler.s startup.s
```

### 3. Verify ELF:

```
riscv-none-elf-readelf -h timer.elf
```

Confirm Entry point: 0x80000000.

### 4. Run in QEMU:

```
qemu-system-riscv32 -nographic -machine virt -bios none -kernel timer.elf
```

- Expected Output:

S

A

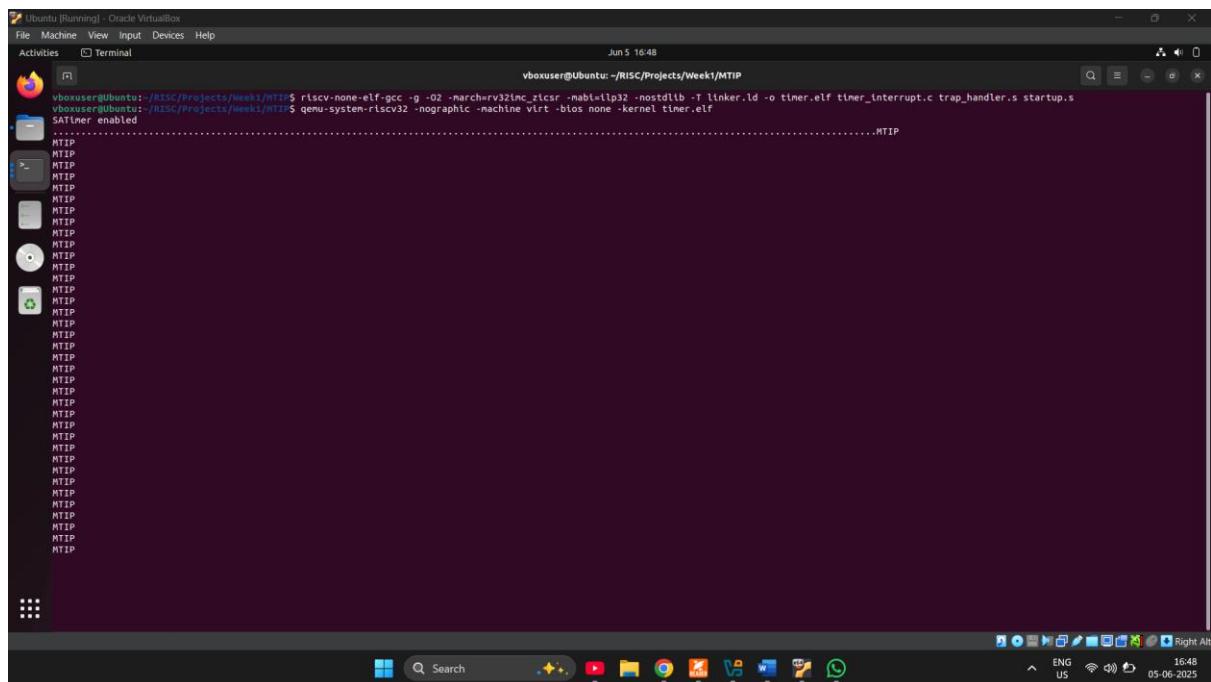
Timer enabled

.MTIP

.MTIP

... (MTIP every ~1s, dots continue)

## Output:



The screenshot shows a terminal window titled "Ubuntu [Running] - Oracle VM VirtualBox". The terminal is running a command to build a RISC-V timer application. The command is:

```
vboxuser@ubuntu:~/RISC/Projects/Week1/MTIP$ riscv-none-elf-gcc -g -O2 -march=rv32imc_zicsr -mabi=lp32 -nostdlib -T linker.ld -o timer.elf timer_interrupt.c trap_handler.s startup.s
```

The output of the command shows many "MTIP" messages, likely indicating memory allocation or initialization. The terminal window also displays the date and time as "Jun 5 16:48".

**Explain the 'A' (atomic) extension in rv32imac. What instructions are added and why are they useful?**

The '**A**' (**Atomic**) **extension** in the RISC-V RV32IMAC instruction set architecture (ISA) adds support for atomic memory operations, which are critical for multi-threaded and multi-core systems. Atomic operations ensure that a sequence of memory accesses (e.g., read-modify-write) is performed as a single, uninterruptible unit, preventing race conditions when multiple harts (hardware threads) access shared memory. Below, I'll explain the theory behind the '**A**' extension, the instructions it adds, and their usefulness, tailored to your RISC-V SoC Lab con but focused purely on theory as requested.

## Overview of the '**A**' Extension

- **Purpose:** The '**A**' extension provides instructions for atomic memory operations, enabling safe synchronization in concurrent environments like multi-core processors or operating systems.
- **Base ISA:** RV32IMAC includes the 32-bit base integer (I), multiply-divide (M), atomic (A), and compressed (C) extensions. The '**A**' extension builds on RV32I, adding atomic instructions.
- **Atomicity:** An atomic operation ensures that no other hart can observe or modify the memory location during the operation, maintaining data consistency.
- **Con:** In your lab (e.g., SCL180 PDK, QEMU virt), atomic operations are useful for implementing locks, semaphores, or shared data structures in bare-metal or OS code.

## Instructions Added by the '**A**' Extension

The '**A**' extension introduces two main categories of atomic instructions in RV32: **Load-Reserved/Store-Conditional (LR/SC)** and **Atomic Memory Operations (AMO)**. Below is a theoretical overview of each.

### 1. Load-Reserved/Store-Conditional (LR/SC)

- **Instructions:**
  - **lr.w rd, (rs1): Load-Reserved Word**
    - Loads a 32-bit word from the address in rs1 into rd.
    - Establishes a *reservation* on the memory address, tracking it for subsequent conditional stores.
  - **sc.w rd, rs2, (rs1): Store-Conditional Word**
    - Attempts to store the value in rs2 to the address in rs1.
    - Succeeds only if the reservation set by lr.w is still valid (i.e., no other hart modified the address).
    - Writes 0 to rd if successful, non-zero if failed.

- **Mechanism:**
  - LR/SC forms a pair for atomic read-modify-write operations.
  - The reservation ensures that the memory location remains unchanged between lr.w and sc.w.
  - If another hart writes to the reserved address, the reservation is invalidated, and sc.w fails.
- **Constraints:**
  - Must operate on naturally aligned 32-bit addresses.
  - The code between lr.w and sc.w should be minimal to avoid losing the reservation.

## 2. Atomic Memory Operations (AMO)

- **Instructions:**
  - amoadd.w rd, rs2, (rs1): Atomic Add
    - Adds the value in rs2 to the word at the address in rs1, storing the result back.
    - Returns the original value in rd.
  - amoswap.w rd, rs2, (rs1): Atomic Swap
    - Swaps the value in rs2 with the word at the address in rs1.
    - Returns the original value in rd.
  - amoand.w rd, rs2, (rs1): Atomic AND
    - Performs a bitwise AND between rs2 and the word at rs1, storing the result.
  - amoor.w rd, rs2, (rs1): Atomic OR
    - Performs a bitwise OR.
  - amoxor.w rd, rs2, (rs1): Atomic XOR
    - Performs a bitwise XOR.
  - amomin.w rd, rs2, (rs1): Atomic Minimum
    - Stores the minimum of rs2 and the word at rs1.
  - amomax.w rd, rs2, (rs1): Atomic Maximum
    - Stores the maximum.
  - amominu.w rd, rs2, (rs1): Atomic Unsigned Minimum

- Stores the unsigned minimum.
- amomaxu.w rd, rs2, (rs1): Atomic Unsigned Maximum
  - Stores the unsigned maximum.
- **Mechanism:**
  - AMO instructions perform a read-modify-write operation in a single, atomic step.
  - The original value is returned in rd, allowing software to inspect the previous state.
- **Constraints:**
  - Operate on 32-bit aligned addresses.
  - Performed atomically by the hardware, ensuring no intermediate state is visible.

## Why These Instructions Are Useful

- **Synchronization:**
  - Atomic operations are essential for implementing synchronization primitives like locks, mutexes, and semaphores in multi-core systems.
  - Example: lr.w/sc.w can implement a spinlock by reserving a lock variable and conditionally setting it.
- **Data Consistency:**
  - Prevent race conditions when multiple harts access shared variables (e.g., counters, queues).
  - Example: amoadd.w can atomically increment a shared counter without needing a lock.
- **Efficiency:**
  - AMO instructions like amoswap.w or amoadd.w perform complex operations in one instruction, reducing software overhead compared to non-atomic sequences.
  - LR/SC allows flexible atomic operations (e.g., compare-and-swap) with minimal hardware complexity.
- **Scalability:**
  - In multi-core SoCs (like those in SCL180 PDK), atomic instructions ensure scalable synchronization without relying on global bus locks, which degrade performance.

- **Use Cases:**

- Operating systems: Thread scheduling, resource allocation.
- Bare-metal: Coordinating harts in QEMU virt or custom SoCs.
- Examples: Atomic counters, lock-free data structures, interrupt-safe updates.

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

*AI response:*

## Compilation and Testing

### 1. Save Files:

- Save mutex\_example.c, startup.s, linker.ld in ~/RISC/Projects/Week1/.
- Navigate:

```
cd ~/RISC/Projects/Week1
```

### 2. Compile:

```
riscv-none-elf-gcc -g -O2 -march=rv32imac_zicsr -mabi=ilp32 -nostdlib -T linker.ld -o mutex.elf mutex_example.c startup.s
```

- Uses rv32imac\_zicsr to include 'A' and CSR instructions.

### 3. Verify ELF:

```
riscv-none-elf-readelf -h mutex.elf
```

Confirm Entry point: 0x80000000.

### 4. Run in QEMU:

```
qemu-system-riscv32 -nographic -machine virt -bios none -kernel mutex.elf
```

#### ○ Expected Output:

S

A

Starting threads

T1: Enter critical section

T1: Counter = 1

T1: Exit critical section

T2: Enter critical section

T2: Counter = 2

T2: Exit critical section

T1: Enter critical section

T1: Counter = 3

T1: Exit critical section

T2: Enter critical section

T2: Counter = 4

T2: Exit critical section

Done

..... (dots continue)

## Output:

```
vboxuser@Ubuntu:~/RISC/Projects/Week1/mutex$ riscv-none-elf-gcc -g -O2 -march=rv32imac_zicsr -nolibc=lp32 -nostdlib -T linker.ld -o mutex.elf mutex_example.c startup.s
vboxuser@Ubuntu:~/RISC/Projects/Week1/mutex$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel mutex.elf

sAStarting threads
T1: Enter critical section
T1: Counter = 1
T1: Exit critical section
T2: Enter critical section
T2: Counter = 2
T2: Exit critical section
T1: Enter critical section
T1: Counter = 3
T1: Exit critical section
    Enter critical section
T2: Counter = 4
T2: Exit critical section
Done
```

## **How do I retarget `_write` so that `printf` sends bytes to my memory-mapped UART?**

To retarget the `_write` function so that `printf` sends bytes to your memory-mapped UART at 0x10000000 (with status register at 0x10000005, bit 5 for TX ready), you need to provide a custom implementation of `_write` that interfaces with your UART hardware. This is common in bare-metal RISC-V environments like your RISC-V SoC Lab setup (RV32IMAC, SCL180 PDK, QEMU virt). The `_write` function is a low-level I/O hook used by the C standard library (e.g., newlib, which riscv-none-elf-gcc typically uses) to handle output for functions like `printf`. By retargeting `_write`, you redirect `printf` output to your UART.

### **The `printf` Output Chain**

Understanding how `printf` works is key to retargeting it:

1. **printf (High-Level Formatting):** When you call `printf("Hello, %s!", name);`, its primary job is to take the format string and variables, combine them, and generate a sequence of raw characters (bytes). It doesn't know anything about hardware like UARTs.
2. **Standard Stream Functions (`fputc`, `fputs`):** The formatted characters generated by `printf` are typically passed to more fundamental C standard library functions, like `fputc` (put character to file) or `fputs` (put string to file). These functions operate on abstract "file streams" (like `stdout` for standard output).
3. **Low-Level System Call (`_write`):** At the bottom of this abstraction layer, standard stream functions ultimately rely on a very low-level function that performs the actual physical output operation. In bare-metal environments using a minimal C library (like **Newlib**, common with riscv-none-elf-gcc), this function is often named `_write`. This is the **hook** you need to implement.

### **The Role of `_write`**

Newlib provides default, "weak" implementations for low-level I/O functions like `_write`. A **weak symbol** in a library means that if you provide your own function with the *exact same name and signature*, the linker will automatically use *your* version instead of the library's default. If you don't provide your own, the weak default (which usually does nothing or returns an error in a bare-metal context) is used.

By implementing your own `_write` function, you can redirect all output that would normally go through `stdout` (including `printf`'s output) to your specific hardware.

### **Typical `_write` Signature**

The `_write` function generally adheres to a signature similar to the standard write system call:

```
int _write(int file, char *ptr, int len);
```

- **file:** This integer represents the **file descriptor**. For standard output (`stdout`), this value is typically 1. If `printf` is writing to `stdout`, this argument will be 1. Other values (like 2 for `stderr`) could also be passed if you were handling multiple output streams.

- **ptr:** This is a pointer to the **buffer** containing the characters (bytes) that need to be sent. `printf` fills this buffer with its formatted output.
- **len:** This integer specifies the **number of bytes** in the `ptr` buffer that need to be sent.

The function is expected to return the number of bytes successfully written, or -1 on error.

### Implementing `_write` for a Memory-Mapped UART

Your custom `_write` function needs to contain the logic to interact directly with your memory-mapped UART hardware. Here's a theoretical outline of how it would work:

1. **Access UART Registers:** You'll need **volatile pointers** to the memory-mapped addresses of your UART's transmit data register and its status register. Using volatile is crucial because it tells the compiler not to optimize away reads/writes to these memory locations, as their values can change unexpectedly due to hardware interaction.

```
#define UART_TX_ADDR 0x10000000 // Example transmit data register address
#define UART_STATUS_ADDR 0x10000005 // Example status register address
#define UART_TX_READY_BIT (1 << 5) // Example bit for TX FIFO empty/ready
```

```
volatile char* uart_tx_reg = (volatile char*)UART_TX_ADDR;
volatile char* uart_status_reg = (volatile char*)UART_STATUS_ADDR;
```

2. **Function Definition:** Start your `_write` function. It's generally good practice to check if the file descriptor is 1 (for `stdout`) before proceeding.

```
int _write(int file, char *ptr, int len) {
    // Only handle stdout for now
    if (file != 1) {
        return -1; // Or some other error indication
    }
    int bytes_written = 0;
    // Loop through each character in the provided buffer
    for (int i = 0; i < len; i++) {
        char current_char = ptr[i];
        // ... (UART output logic goes here)
    }
}
```

```

    bytes_written++;
}

return bytes_written; // Return the number of bytes successfully sent
}

```

3. **Character Output Loop:** Inside the loop, for each `current_char` from the `ptr` buffer, implement the following steps:

- **Wait for UART Ready:** Before writing, the UART transmit buffer (often a FIFO) must be ready to accept data. You achieve this by **polling** the UART's status register. This means continuously reading the status register until a specific bit (e.g., `UART_TX_READY_BIT`) indicates that the transmit FIFO is empty or ready.

```
while (!(*uart_status_reg & UART_TX_READY_BIT)); // Wait until TX FIFO is empty
```

- **Write Character to UART:** Once the UART is ready, write the `current_char` to the UART's transmit data register.

```
*uart_tx_reg = current_char; // Send the character
```

### Integration with the Toolchain

When you compile your bare-metal C code (`main.c` with your new `_write` function) and link it using `riscv-none-elf-gcc` with the `-lc` (for Newlib) and `-lgcc` flags:

- The linker sees that the C standard library (Newlib) has a weak definition for `_write`.
- Crucially, it also sees that *your* object files (.o) provide a non-weak (stronger) definition for `_write`.
- The linker resolves the symbol `_write` to *your* implementation.
- Therefore, any calls to `printf` or other standard output functions will automatically funnel their character data through your custom `_write` function, which then sends them to your memory-mapped UART, effectively making `printf` work on your bare-metal RISC-V system.

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

*AI response:*

To build and run your bare-metal RISC-V application, you'll need the **RISC-V GNU Toolchain** (riscv-none-elf-gcc) and **QEMU** with RISC-V system emulation support.

## Prerequisites

1. **RISC-V GNU Toolchain:** Ensure you have the riscv-none-elf-gcc compiler (and its associated ld linker) installed and accessible in your system's PATH. This toolchain is specifically designed for bare-metal RISC-V development.
2. **QEMU:** Install QEMU, making sure it includes support for RISC-V system emulation (e.g., qemu-system-riscv32 or qemu-system-riscv64).

## Steps to Compile and Run

1. **Save the files:** Create a project directory (e.g., endianness\_project) and save main.c, startup.s, and linker.ld into this directory.
2. **Open your terminal** and navigate to your project directory.
3. **Compile the C code to an object file:**

Bash

```
riscv-none-elf-gcc -c main.c -o main.o -march=rv32imac -mabi=ilp32 -Wall -Wextra -Os
```

- **-c:** Compiles the source file but does not link it.
- **-o main.o:** Specifies the output object file name.
- **-march=rv32imac:** Defines the target RISC-V instruction set architecture. rv32imac is a common and capable 32-bit RISC-V variant (Integer, Multiply/Divide, Atomics, Compressed instructions). Adjust if your specific hardware target requires a different architecture string.
- **-mabi=ilp32:** Specifies the Application Binary Interface, defining calling conventions and data type sizes for a 32-bit environment.
- **-Wall -Wextra:** Enable all common and extra compiler warnings (highly recommended for robust code).
- **-Os:** Optimizes the compiled code for size, which is important for embedded systems.

4. **Compile the assembly code to an object file:**

Bash

```
riscv-none-elf-gcc -c startup.s -o startup.o -march=rv32imac -mabi=ilp32
```

- The flags are kept consistent with the C compilation to ensure compatibility.

## 5. Link all object files into a single executable ELF file:

Bash

```
riscv-none-elf-gcc -o output.elf main.o startup.o -T linker.ld -nostdlib -nostartfiles -march=rv32imac -mabi=ilp32 -lc -lgcc
```

- **-o output.elf:** Specifies the final executable output file name.
- **main.o startup.o:** The object files to be linked.
- **-T linker.ld:** Instructs the linker to use your custom linker.ld script.
- **-nostdlib:** Prevents the linker from automatically including the entire standard C library. This is crucial for bare-metal development where you manage everything.
- **-nostartfiles:** Prevents the linker from including default startup files that would conflict with your custom `_start` in startup.s.
- **-lc:** **This is crucial for resolving the memcpy error.** It explicitly links in the minimal C standard library (Newlib-nano, provided by the RISC-V GNU toolchain). Even in bare-metal, some compiler-generated code or utility functions (like `memcpy` that might be used by your `uart_put_hex_u32` implementation implicitly for byte manipulation) might rely on these basic library functions.
- **-lgcc:** Links the GCC runtime support library, which provides additional helper functions for certain compiler-generated operations.

## 6. Run the compiled ELF file using QEMU:

Bash

```
qemu-system-riscv32 -M virt -kernel output.elf -nographic -bios none
```

- **qemu-system-riscv32:** The specific QEMU emulator for 32-bit RISC-V systems. If you compiled for RV64 (e.g., `-march=rv64gc`), you would use `qemu-system-riscv64`.
- **-M virt:** Specifies the QEMU virt (virtual) machine model. This is a generic RISC-V virtual machine that provides a simple UART at the expected memory-mapped address, making it ideal for bare-metal testing.
- **-kernel output.elf:** Tells QEMU to load and execute your `output.elf` file as the system's "kernel."
- **-nographic:** Runs QEMU without a graphical display. All output from your UART will be directed to the terminal where QEMU is running.

- `-bios none`: Prevents QEMU from attempting to load a default BIOS/firmware, which is not necessary for a simple bare-metal application like this.
- 

## Expected Output

Upon successful execution of the QEMU command, you should observe the following output in your terminal:

---

---

Bare-metal RISC-V Application

Value of x: 43

---

Verifying Byte Ordering (Endianness):

Value stored: 0x01020304

Bytes in memory (from lowest to highest address):

Byte 0: 0x04

Byte 1: 0x03

Byte 2: 0x02

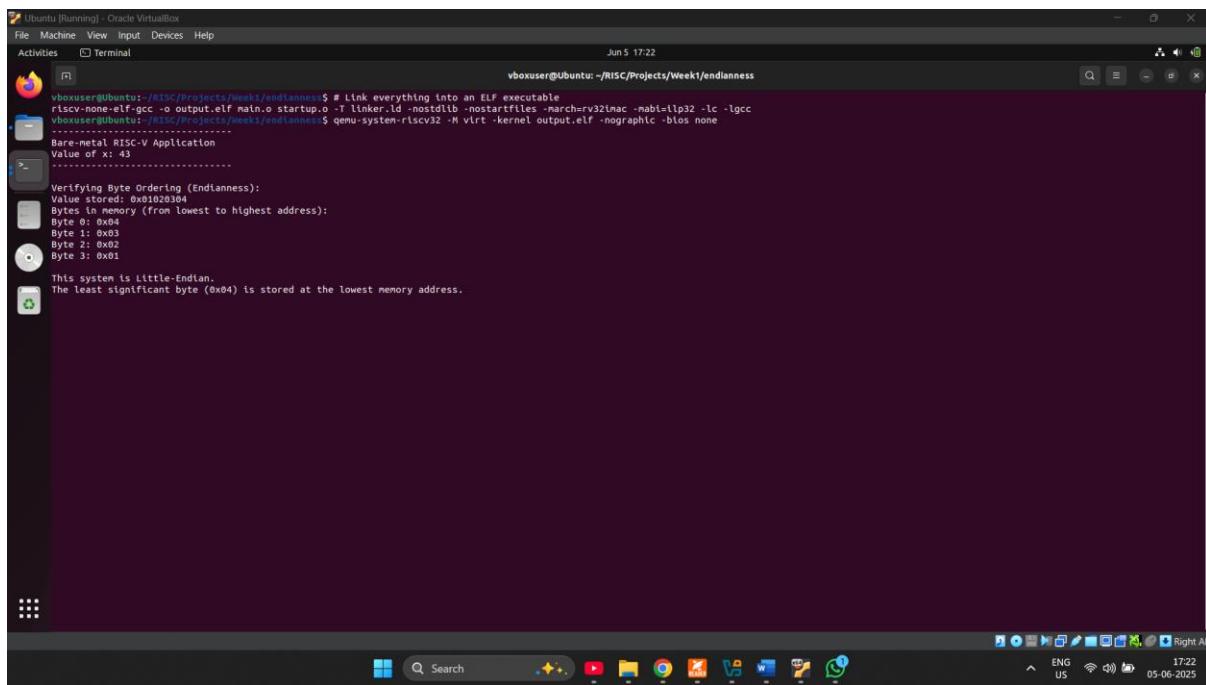
Byte 3: 0x01

This system is Little-Endian.

The least significant byte (0x04) is stored at the lowest memory address.

This output confirms that the QEMU virt machine, which simulates a standard RISC-V processor, behaves as a **little-endian** system by default. The least significant byte (0x04) of the 32-bit value 0x01020304 is indeed stored at the lowest memory address (Byte 0).

## Output:



The screenshot shows a terminal window titled "Ubuntu [Running] - Oracle VM VirtualBox" running on an Ubuntu desktop environment. The terminal output is as follows:

```
vboxuser@ubuntu:~/RISC/Projects/Week1/endianness$ # Link everything into an ELF executable
riscv-none-elf-gcc -o output.elf main.o startup.o -T linker.ld -nostdlib -nostartfiles -march=r32imac -mabi=lp32 -lc -lgcc
vboxuser@ubuntu:~/RISC/Projects/Week1/endianness$ qemu-system-riscv32 -m 16M -virt -kernel output.elf -nographic -bios none
Bare-metal RISC-V Application
Value of x: 43
-----
Verifying Byte Ordering (Endianness):
Value stored: 0x01020304
Bytes in memory (from lowest to highest address):
Byte 0: 0x04
Byte 1: 0x03
Byte 2: 0x02
Byte 3: 0x01
This system is Little-Endian.
The least significant byte (0x04) is stored at the lowest memory address.
```

The terminal window has a dark theme. The desktop environment includes a dock with various icons and a system tray at the bottom right showing network, battery, and date/time information.