

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?"

Ans:

1. Unpack the Toolchain

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

2. Add to PATH

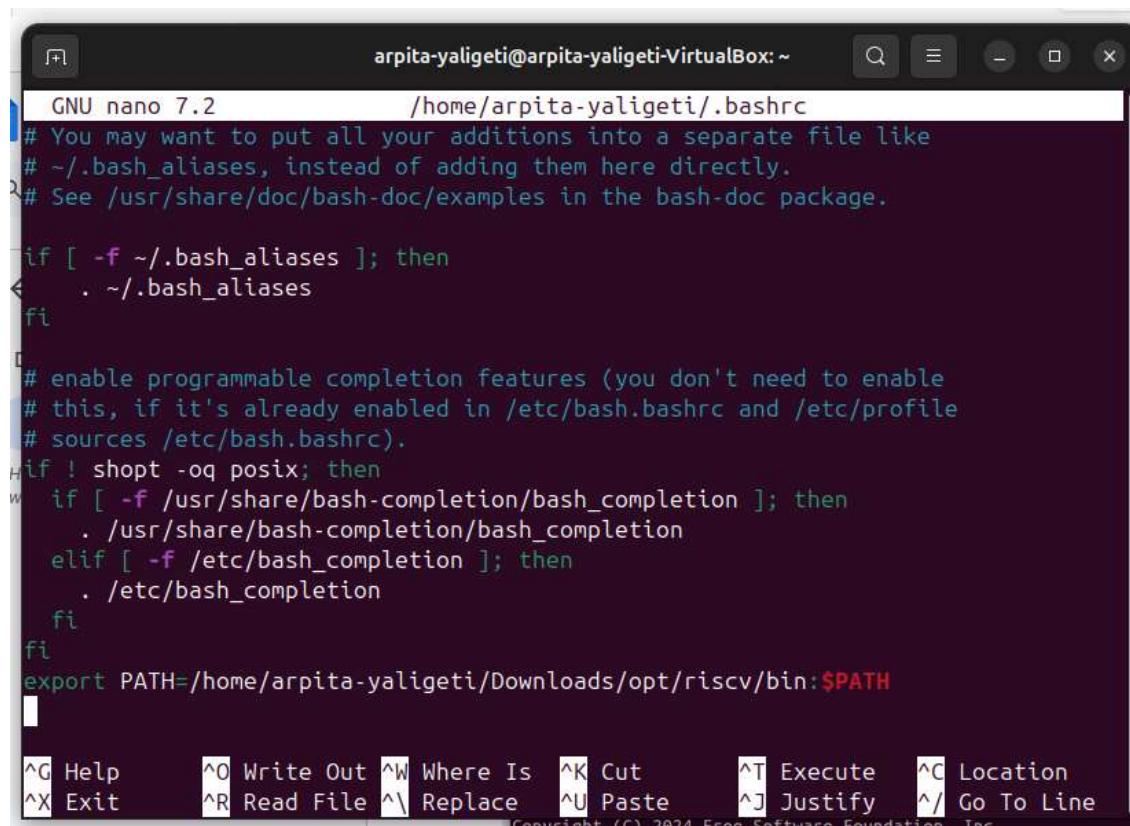
```
nano ~/.bashrc
```

```
export PATH=home/arpita_yaligeti/riscv/bin:
```

```
source ~/.bashrc
```

3. Sanity Check / Verify Installation

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



The screenshot shows a terminal window titled "arpita-yaligeti@arpita-yaligeti-VirtualBox: ~". The window contains the command "GNU nano 7.2 /home/arpita-yaligeti/.bashrc". The text in the editor is as follows:

```
GNU nano 7.2          /home/arpita-yaligeti/.bashrc
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
    if [ -f /usr/share/bash-completion/bash_completion ]; then
        . /usr/share/bash-completion/bash_completion
    elif [ -f /etc/bash_completion ]; then
        . /etc/bash_completion
    fi
fi
export PATH=/home/arpita-yaligeti/Downloads/opt/riscv/bin:$PATH
```

The bottom of the terminal shows the nano key bindings:

^G	Help	^O	Write Out	^W	Where Is	^K	Cut	^T	Execute	^C	Location
^X	Exit	^R	Read File	^V	Replace	^U	Paste	^J	Justify	^/	Go To Line

```
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ tar -xzf riscv-toolchain-rv32imac-x86_64-ubuntu.tar.gz
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano ~/.bashrc
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ source ~/.bashrc
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ riscv32-unknown-elf-gcc --version
riscv32-unknown-elf-objdump --version
riscv32-unknown-elf-gdb --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.

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.

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.
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$
```

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

Ans: 1. create a file named hello.c:

```
#include <stdio.h>

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

```
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano hello.c
GNU nano 7.2
#include <stdio.h>

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

2. Use the RISC-V GCC toolchain to compile it into an ELF binary:

```
riscv32-unknown-elf-gcc -o hello.elf hello.c
```

3. Check the architecture of the generated binary:

file hello.elf

3 From C to Assembly

Question

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

Ans:

1. Run the following command:

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

```
arpita-yaligeti@arpita-yaligeti-VirtualBox: ~
```

```
arpita-yaligeti@arpita-yaligeti-VirtualBox: $ riscv32-unknown-elf-gcc -S -O0 hello.c
```

```
arpita-yaligeti@arpita-yaligeti-VirtualBox: $ riscv32-unknown-elf-gcc -S -O0 hello.c
```

```
arpita-yaligeti@arpita-yaligeti-VirtualBox: $
```

2. Prologue & Epilogue Explanation

In the hello.s file, inside the main: label, you will typically see:

hello.c

hello.s

x

```
.file  "hello.c"
.option nopic
.attribute arch, "rv32i2p1_m2p0_a2p1_c2p0"
.attribute unaligned_access, 0
.attribute stack_align, 16
.text
.section      .rodata
.align 2

.LC0:
.string "Hello, world!"
.text
.align 1
.globl main
.type  main, @function

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)
    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
```

Start of main:

sp (stack pointer) is moved down to allocate space on the stack.

The function saves ra (return address) and s0 (frame pointer) so they can be restored before returning.

s0 becomes the base for accessing local variables and arguments.

End of main:

Restores the original state of the stack and registers.

ret uses the restored ra to jump back to the caller (usually the startup code or exit handler).

The prologue/epilogue manages: Stack space

Callee-saved registers

Clean return to the caller

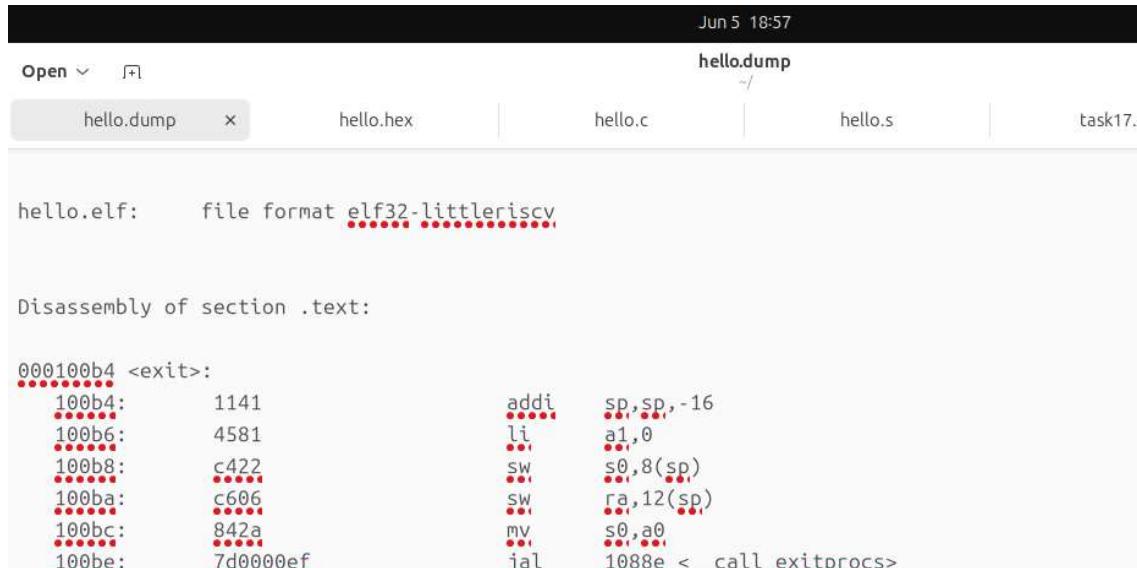
This is required by the RISC-V calling convention, even in simple programs.

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

Ans: 1: Disassemble the ELF File: Use objdump to generate a readable disassembly:

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

This creates a human-readable file hello.dump with the disassembled instructions from the ELF.



The screenshot shows the GDB interface with the assembly dump of the hello.elf file. The assembly dump shows the following code:

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

Disassembly of section .text:
000100b4 <exit>:
100b4:    1141          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>
```

Jun 5 18:58

hello.dump

hello.dump	x	hello.hex	hello.c	hello.s	task17.c
100dc:	842d		mv	sv,d0	
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		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>:					
100e2:	00004197		auipc	gp,0x4	
100e6:	b9e18193		addi	gp, gp,-1122 # 13c80 <__global_pointer\$>	
100ea:	d4818513		addi	a0, gp,-696 # 139c8 <__stdio_exit_handler>	
100ee:	07018613		addi	a2, gp,112 # 13cf0 <__BSS_END__>	
100f2:	8e09		sub	a2,a2,a0	
100f4:	4581		li	a1,0	
100f6:	2571		jal	10782 <memset>	
100f8:	00001517		auipc	a0,0x1	
100fc:	86450513		addi	a0,a0,-1948 # 1095c <atexit>	
10100:	c519		beqz	a0,1010e <start+0x2c>	
10102:	00002517		auipc	a0,0x2	
10106:	90c50513		addi	a0,a0,-1780 # 11a0e <__libc_fini_array>	
1010a:	053000ef		jal	1095c <atexit>	
1010e:	2529		jal	10718 <__libc_init_array>	
10110:	4502		lw	a0,0(sp)	
10112:	004c		addi	a1,sp,4	
10114:	4601		li	a2,0	
10116:	20b1		jal	10162 <main>	
10118:	bf71		j	100b4 <exit>	
0001011a <__do_global_dtors_aux>:					
1011a:	1141		addi	sp,sp,-16	
1011c:	c422		sw	s0,8(sp)	
1011e:	d641c783		lbu	a5,-668(gp) # 139e4 <completed.1>	
10122:	c606		sw	ra,12(sp)	

Jun 5 18:59

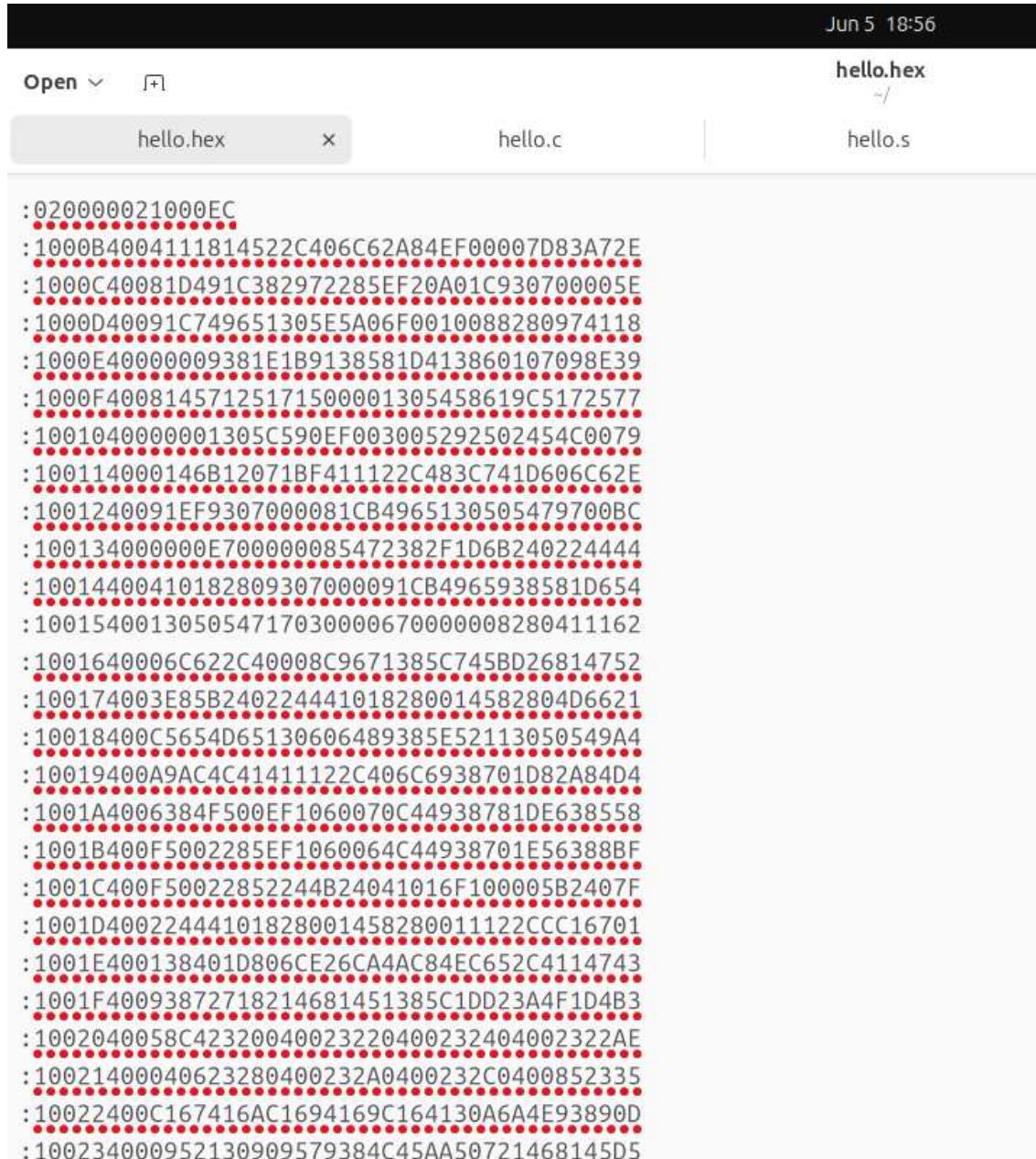
hello.dump

hello.dump	x	hello.hex	hello.c	hello.s	task17.i
100f8:	00001517		auipc	a0,0x1	
100fc:	86450513		addi	a0,a0,-1948 # 1095c <atexit>	
10100:	c519		beqz	a0,1010e <start+0x2c>	
10102:	00002517		auipc	a0,0x2	
10106:	90c50513		addi	a0,a0,-1780 # 11a0e <__libc_fini_array>	
1010a:	053000ef		jal	1095c <atexit>	
1010e:	2529		jal	10718 <__libc_init_array>	
10110:	4502		lw	a0,0(sp)	
10112:	004c		addi	a1,sp,4	
10114:	4601		li	a2,0	
10116:	20b1		jal	10162 <main>	
10118:	bf71		j	100b4 <exit>	
0001011a <__do_global_dtors_aux>:					
1011a:	1141		addi	sp,sp,-16	
1011c:	c422		sw	s0,8(sp)	
1011e:	d641c783		lbu	a5,-668(gp) # 139e4 <completed.1>	
10122:	c606		sw	ra,12(sp)	

2: Convert ELF to Intel HEX Format: Use objcopy to turn the ELF into a raw hex dump:

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

This creates hello.hex, a file in Intel HEX format, often used for loading onto microcontrollers or simulators.



The screenshot shows a terminal window with the following details:

- Top right: Jun 5 18:56
- File list:
 - Open dropdown
 - hello.hex (selected)
 - hello.c
 - hello.s
- Content pane:

```
:020000021000EC
:1000B4004111814522C406C62A84EF00007D83A72E
:1000C40081D491C382972285EF20A01C930700005E
:1000D40091C749651305E5A06F0010088280974118
:1000E40000009381E1B9138581D413860107098E39
:1000F40081457125171500001305458619C5172577
:1001040000001305C590EF003005292502454C0079
:100114000146B12071BF411122C483C741D606C62E
:1001240091EF9307000081CB4965130505479700BC
:100134000000E700000085472382F1D6B240224444
:10014400410182809307000091CB4965938581D654
:10015400130505471703000670000008280411162
:1001640006C622C40008C9671385C745BD26814752
:100174003E85B240224441018280014582804D6621
:10018400C5654D65130606489385E52113050549A4
:10019400A9AC4C41411122C406C6938701D82A84D4
:1001A4006384F500EF1060070C44938781DE638558
:1001B400F5002285EF1060064C44938701E56388BF
:1001C400F50022852244B24041016F100005B2407F
:1001D40022444101828001458280011122CCC16701
:1001E400138401D806CE26CA4AC84EC652C4114743
:1001F40093872718214681451385C1DD23A4F1D4B3
:1002040058C42320040023220400232404002322AE
:10021400040623280400232A0400232C0400852335
:10022400C167416AC1694169C164130A6A4E93890D
:100234000952130909579384C45AA50721468145D5
```

3. Explanation: A typical line in hello.dump might look like:

00010074 <main>:

10074: 1141 addi sp, sp, -16
10076: c606 sw ra, 12(sp)

10074: Address — memory address of the instruction

1141 Opcode — raw machine code (hex)

addi sp, sp, -16 Mnemonic + Operands — human-readable instruction

Address Where in memory the instruction is located

Opcode The actual binary (in hex) form of the instruction

Mnemonic The assembly name of the operation (e.g., addi, sw, ret, etc.)

Operands Registers or immediates the instruction works with (e.g., sp, ra)

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

Roles

Ans:

RV32I Register Table

Number	Name	ABI Name	Description / Role
x0	zero	zero	Constant 0 (hard-wired zero)
x1	ra	Return Address	Used to return from function calls
x2	sp	Stack Pointer	Points to top of stack
x3	gp	Global Pointer	Points to global data
x4	tp	Thread Pointer	Thread-local storage
x5	t0	Temporary	Caller-saved temporary
x6	t1	Temporary	Caller-saved temporary
x7	t2	Temporary	Caller-saved temporary
x8	s0	Saved Register / Frame Pointer	Callee-saved, often frame pointer
x9	s1	Saved Register	Callee-saved
x10	a0	Argument / Return Value	First argument, or return value

x11	a1	Argument / Return Value	Second argument, or return value
x12	a2	Argument	3rd argument
x13	a3	Argument	4th argument
x14	a4	Argument	5th argument
x15	a5	Argument	6th argument
x16	a6	Argument	7th argument
x17	a7	Argument	8th argument
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 temporary
x29	t4	Temporary	Caller-saved temporary
x30	t5	Temporary	Caller-saved temporary
x31	t6	Temporary	Caller-saved temporary

RISC-V Calling Convention Summary

a0–a7 (x10–x17): Function arguments and return values

s0–s11 (x8–x9, x18–x27): Callee-saved registers (must be preserved by the function)

t0–t6 (x5–x7, x28–x31): Caller-saved temporaries (not preserved across function calls)

ra (x1): Return address for ret

sp (x2): Stack pointer — managed by each function's prologue/epilogue

s0 (x8): Often doubles as frame pointer

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

Ans:

1. Minimal C program

```
int main() {
    volatile int a = 5;
    volatile int b = 6;
    volatile int c = a + b;
    return 0;
}
```

Don't use printf() — GDB's simulator has no support for system calls or stdout.



The screenshot shows a terminal window with three tabs. The active tab is titled "GNU nano 7.2" and contains the following C code:

```
int main() {
    volatile int a = 5;
    volatile int b = 6;
    volatile int c = a + b;
    return 0;
}
```

2. Load and run in GDB

riscv32-unknown-elf-gdb hello.elf

Then inside GDB:

```
(gdb) target sim
(gdb) load
(gdb) break main
(gdb) run
```

```

arpita-yaligeti@arpita-yaligeti-VirtualBox: ~      arpita-yaligeti@arpita-yaligeti-VirtualBox: ~      arpita-yaligeti@arpita-yaligeti-VirtualBox: ~
arpita-yaligeti@arpita-yaligeti-VirtualBox: $ riscv32-unknown-elf-gcc -march=rv32i -mabi=ilp32 -nostdlib -o hello.elf hello.c
/home/arpita-yaligeti/Downloads/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/14.2.0/../../../../riscv32-unknown-elf/bin/ld: warning: cannot find entry symbol _start; defaulting to 00010094
arpita-yaligeti@arpita-yaligeti-VirtualBox: $ 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...
(No debugging symbols found in hello.elf)
(gdb) target sim
Connected to the simulator.
(gdb) load
Loading section .text, size 0x48 lma 10094
Start address 10094
Transfer rate: 576 bits in <1 sec.
(gdb) break main
Breakpoint 1 at 0x100d8
(gdb) run

Breakpoint 1 at 0x100d8
(gdb) run
Starting program: /home/arpita-yaligeti/hello.elf

Breakpoint 1, 0x000100d8 in main ()

```

7. Give me spike or QEMU commands to boot my bare-metal ELF and print to the 'UART' console. Give me spike or QEMU commands to boot my bare-metal ELF and print to the 'UART' console.

1: Write a Minimal Bare-Metal UART Program

hello2.c

```

#define UART0 0x10000000

void uart_putc(char c) {
    *(volatile char *)UART0 = c;
}

void uart_puts(const char *s) {
    while (*s) uart_putc(*s++);
}

int main() {
    uart_puts("Printing from UART\n");
    uart_puts("Value of x: 43\n");
    while (1);
}

```

2: Add a Startup File

startup.s

```
.section .init
.globl _start

_start:
    la sp, _stack_top
    call main
1: j 1b
```

```
.section .bss
.space 1024
_stack_top:
```

3: Write a Linker Script (linker.ld)

ENTRY(_start)

```
MEMORY {
    FLASH (rx) : ORIGIN = 0x80000000, LENGTH = 512K
    RAM  (rwx): ORIGIN = 0x80080000, LENGTH = 512K
}

SECTIONS {
    .text : {
        *(.init)
        *(.text*)
        *(.rodata*)
    } > FLASH

    .data : {
        *(.data*)
    } > RAM

    .bss : {
        *(.bss*)
        *(COMMON)
    } > RAM
}
```

4: Compile the Program

```
riscv32-unknown-elf-gcc -g -march=rv32im -mabi=ilp32 -nostdlib \-T linker.ld -o hello2.elf
hello2.c startup.s
```

-nostdlib: Avoid linking with newlib.

-T linker.ld: Use your custom memory layout

4: Compile the Program

5: Verify the ELF File

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

Check: Machine: RISC-V

Class: ELF32

Entry point: 0x80000000

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

Ensure: .text in FLASH with R E

.data, .bss in RAM with R W

6: Run with QEMU and See UART Output

```
qemu-system-riscv32 \
    -nographic \
    -machine virt \
    -bios none \
    -kernel hello2.elf
```

The screenshot shows a terminal window titled "arpita-yaligeti@arpita-yaligeti-VirtualBox: ~". The terminal output is as follows:

```
Jun 5 19:50
arpita-yaligeti@arpita-yaligeti: ~
arpita-yaligeti@arpita-yaligeti-VirtualBox: $ riscv32-unknown-elf-gcc -g -march=r32im -mabi=ilp32 -nostdlib \ -T linker2
.lds -o hello2.elf hello2.c startup.s
arpita-yaligeti@arpita-yaligeti-VirtualBox: $ 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: 5996 (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
arpita-yaligeti@arpita-yaligeti-VirtualBox: $ riscv32-unknown-elf-readelf -l hello2.elf

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

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

Ans:

1: Use a Simple C Program

```
// hello.c
int add(int x, int y) {
    return x + y;
}

int main() {
    int result = a
    return 0;
}
```

```
GNU nano 7.2
int add(int x, int y) {
    return x + y;
}

int main() {
    int result = add(3, 4);
    return 0;
}
```

2: Generate Assembly

```
riscv32-unknown-elf-gcc -S -O0 hello.c -o hello_O0.s  
riscv32-unknown-elf-gcc -S -O2 hello.c -o hello_O2.s
```

```
Jun 5 21:39
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano eight.c
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ riscv32-unknown-elf-gcc -S -O0 eight.c -o eight_00.s
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ riscv32-unknown-elf-gcc -S -O2 eight.c -o eight_02.s
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ diff -y hello_00.s hello_02.s | less
diff: hello_00.s: No such file or directory
diff: hello_02.s: No such file or directory
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ diff -y eight_00.s eight_02.s | less
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$
```

```
Jun 5 21:38
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$
```

<pre>.file "eight.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: addi sp,sp,-32 sw ra,28(sp) sw s0,24(sp) addi s0,sp,32 sw a0,-20(\$0) sw a1,-24(\$0) lw a4,-20(\$0) lw a5,-24(\$0) 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) :</pre>	<pre>.file "eight.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: addi sp,sp,-32 sw ra,28(sp) sw s0,24(sp) addi s0,sp,32 sw a0,-20(\$0) sw a1,-24(\$0) lw a4,-20(\$0) lw a5,-24(\$0) 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)</pre>	<pre>.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(\$0) 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</pre>
--	--	---

(END)

3: Side-by-Side Summary of Differences

Aspect	-O0 (No Optimization)	-O2 (Optimized)
Function Calls	Calls add() as a separate function	Inline add(3, 4) into main()
Instructions	Many extra moves, stack usage, and saves	Fewer instructions, minimal stack/frame use
Prologue/Epilogue	Full frame setup with sp, s0, etc.	Sometimes omitted entirely
Dead Code	Keeps all assignments (even unused vars)	Eliminates variables that have no side effects
Register Usage	Conservative — uses memory and more lw/sw	Aggressive register allocation (a0, t0, etc.)
Labels/Comments	More verbose for debugging	Cleaner, smaller code

-O0: No optimization

- Prioritizes debugging and clarity
- Keeps stack frames, variable assignments, and separate function calls
- Avoids register reuse to make stepping easier in a debugger

-O2: Aggressive optimization

- Inline simple functions (like add)
- Removes dead code (e.g., unused result)
- Minimizes memory use, allocates registers smartly
- Avoids unnecessary stack usage

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

Ans.

1: Write the C code (rdcycle.c)

```
#include <stdint.h>
```

```
static inline uint32_t rdcycle(void) {
    uint32_t cycles;
    __asm__ volatile ("rdcycle %0" :
    "=r"(cycles));
    return cycles;
}
```

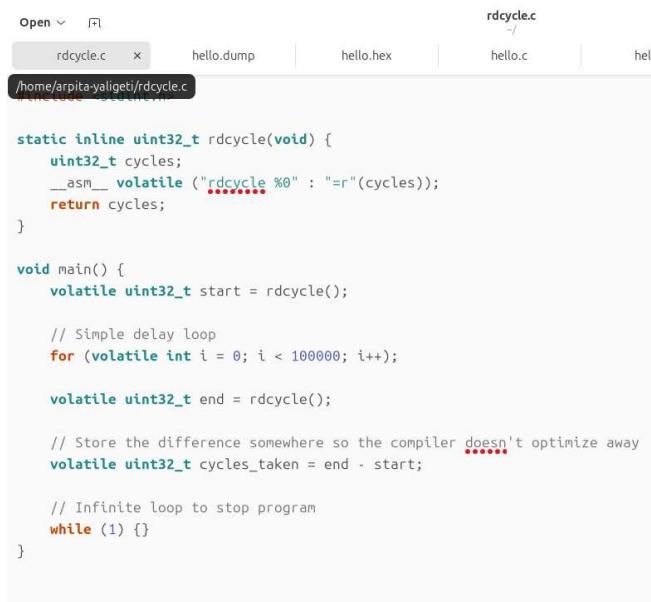
```
void main() {
    volatile uint32_t start = rdcycle();

    // Simple delay loop
    for (volatile int i = 0; i < 100000;
i++) {

        volatile uint32_t end = rdcycle();

        // Calculate cycles taken during
delay
        volatile uint32_t cycles_taken = end - start;

        // Infinite loop to keep program alive
        while (1)
    }
```



```
rdcycle.c x hello.dump | hello.hex | hello.c | hel
/home/arpita-yaligeet/rdcycle.c

static inline uint32_t rdcycle(void) {
    uint32_t cycles;
    __asm__ volatile ("rdcycle %0" : "=r"(cycles));
    return cycles;
}

void main() {
    volatile uint32_t start = rdcycle();

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

    volatile uint32_t end = rdcycle();

    // Store the difference somewhere so the compiler doesn't optimize away
    volatile uint32_t cycles_taken = end - start;

    // Infinite loop to stop program
    while (1) {}
}
```

Explanation: The rdcycle() function uses inline assembly rdcycle %0 to read the cycle counter CSR into a register, then stores it in the C variable cycles.

The constraints "=r"(cycles) tell the compiler to allocate a register for the output and write the result there.

volatile on variables ensures the compiler does not optimize away reads/writes.

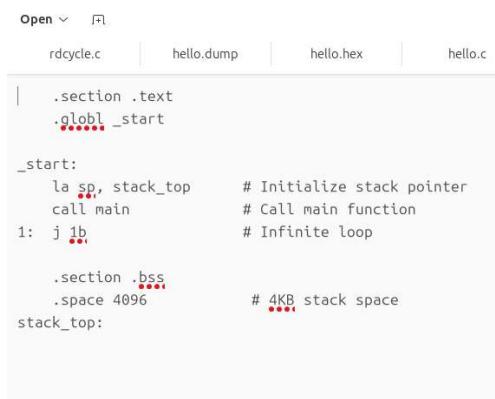
main() measures cycles taken by a delay loop.

The program enters an infinite loop to keep running

2: Create a startup file (startup.s)

```
.section .text
.globl _start

_start:
    la sp, stack_top      # Set up stack pointer
    call main             # Call main function
```



```
Open ▾ rdcycle.c hello.dump hello.hex hello.c

| .section .text
| .globl _start
|
| _start:
|     la sp, stack_top      # Initialize stack pointer
|     call main             # Call main function
|     j 1b                  # Infinite loop
|
| .section .bss
| .space 4096               # 4KB stack space
| stack_top:
```

1: j 1b # Infinite loop to halt

```
.section .bss
.space 4096      # 4 KB stack space
stack_top:
```

3: Create a simple linker script (rdcycle.ld)

```
ENTRY(_start)

SECTIONS
{
    . = 0x80000000;

    .text : {
        *(.text*)
    }

    .bss : {
        *(.bss*)
        . = ALIGN(4);
    }
}
```



4: Compile the program

```
riscv32-unknown-elf-gcc -g -O0 -march=rv32imac -mabi=ilp32 -nostdlib -T rdcycle.ld -o rdcycle.elf rdcycle.c startup.s
```

-nostdlib avoids linking libc (no printf needed).

-T rdcycle.ld specifies the custom linker script.

This produces a bare-metal ELF executable.

5: Run with QEMU

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

Expected Output and Behavior:

No output will be printed because there is no I/O.

QEMU will run your program indefinitely without crashing.

The infinite loop in main() keeps the program alive.

You can confirm it ran successfully because QEMU stays open and stable.

To see cycle count values, you'd need to add debug support (e.g., connect via riscv32-unknown-elf-gdb) or implement UART output.

Why No printf?

Your toolchain lacks rdmon.specs or semihosting support.

Linking with -nostdlib and no standard C library avoids unresolved symbols like printf.

This lets you test inline assembly and bare-metal functionality simply.

Explanation of Inline Assembly Constraints:

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

"rdcycle %0" is the assembly instruction reading CSR 0xC00 (cycle counter) into the output register.

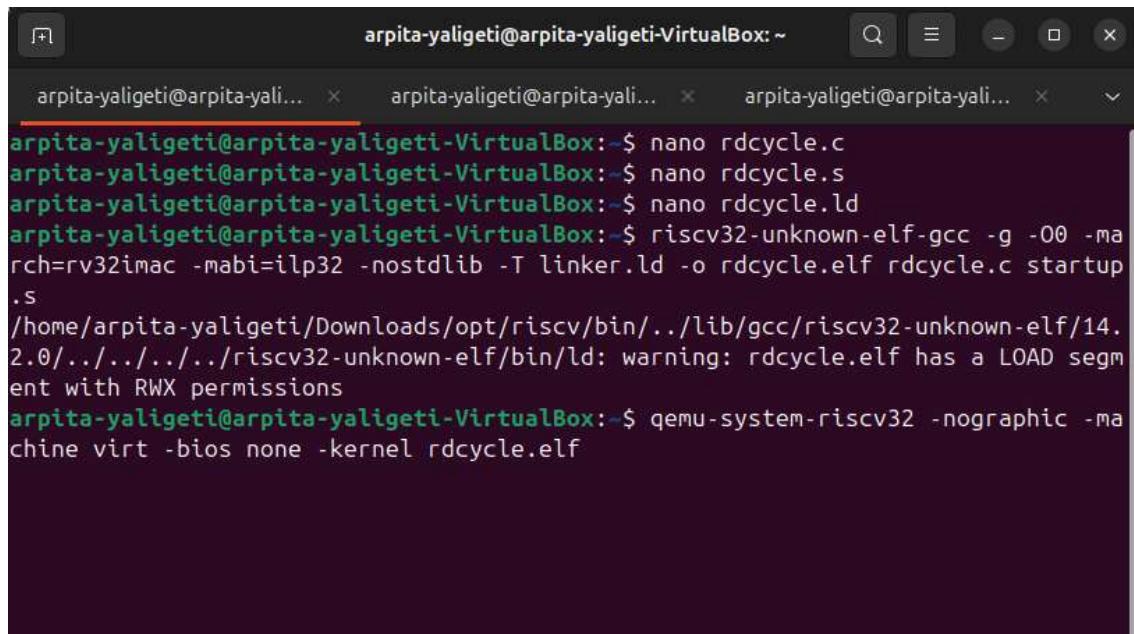
"=r"(cycles) means:

=: write-only operand (output).

r: allocate any general-purpose register.

(cycles): store the result in the C variable cycles.

volatile prevents the compiler from optimizing away or reordering the instruction.



The screenshot shows a terminal window with three tabs open, all showing the same command-line interface. The current tab is active and displays the following sequence of commands and their output:

```
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano rdcycle.c
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano rdcycle.s
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano rdcycle.ld
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ riscv32-unknown-elf-gcc -g -O0 -march=r32imac -mabi=ilp32 -nostdlib -T linker.ld -o rdcycle.elf rdcycle.c startup.s
/home/arpita-yaligeti/Downloads/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/14.2.0/.../.../.../riscv32-unknown-elf/bin/ld: warning: rdcycle.elf has a LOAD segment with RWX permissions
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ qemu-system-riscv32 -nographic -machine virt -bios none -kernel rdcycle.elf
```

1. `asm volatile (...):` `asm` is the GCC keyword for embedding assembly code in C. `volatile` tells the compiler not to optimize away this assembly block, even if the result is unused. Without `volatile`, the compiler might remove the instruction during optimization, thinking it has no side effects.
2. `"csrr %0, cycle"`: This is the assembly template.
`csrr` stands for Control and Status Register Read.
`cycle` is the CSR number `0xC00`, which contains the current CPU cycle count.
`%0` is a placeholder for the output operand (which will be replaced by a register, like `a0`, `t0`, etc.).
3. `: "=r"(c)`: This is the output constraint section of the inline assembly.
`=` → Means the operand is write-only (i.e., the assembly code writes to it).
`r` → Tells GCC to use a general-purpose register to store the result.
`(c)` → Specifies that the result should be stored in the C variable `c`.

Together, `"=r"(c)` tells the compiler:

"Use a register to hold the result of this instruction, and then assign that register's value to the variable `c`."

Summary Table

Element	Meaning
<code>asm</code>	GCC keyword for inline assembly
<code>volatile</code>	Prevents compiler optimizations/removal
<code>"csrr %0, cycle"</code>	Assembly instruction to read cycle CSR into register
<code>%0</code>	Placeholder for the output operand (resolved by GCC to a register)
<code>"=r" (c)</code>	Output constraint: write-only (=), general-purpose register (r), binds to c

Why Use This ?

Efficiently read the hardware cycle counter without C library support.

Perfect for bare-metal, embedded, or performance timing use cases on RISC-V processors.

10 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?

Ans: Code Snippet

```
#include <stdint.h>

int main(void) {
    volatile uint32_t *gpio = (uint32_t *)0x10012000; // volatile to prevent optimization
    *gpio = 0x1; // Write to the GPIO register to set/toggle the pin
    return 0;
}
```



The screenshot shows a terminal window titled 'arpita-yaligeti@arpita-yaligeti-VirtualBox: ~'. The current directory is indicated by three dots. The terminal title bar also shows four other tabs. The main pane of the terminal displays the code for '10.c' in a monospaced font. The code is identical to the one above, with the addition of a comment line: ' // volatile to prevent optimization'.

Why Use volatile?

The volatile keyword tells the compiler:

This variable may change at any time, without the compiler knowing.

So the compiler must:

Always read from or write directly to the memory location.

Avoid optimizing out what might look like a "redundant" access.

Without volatile, the compiler might assume:

```
uint32_t *gpio = (uint32_t *)0x10012000;
*gpio = 0x1;
*gpio = 0x1; // Might optimize away this second write
```

Because it sees the same value being written twice — not realizing it's a hardware register.

Alignment Consideration

The `uint32_t *` type implies 4-byte (32-bit) access alignment. This means:

The address `0x10012000` must be aligned on a 4-byte boundary.

That address is aligned (`0x10012000 % 4 == 0`), so it's safe.

If the address were unaligned (e.g., `0x10012001`), a 32-bit write could cause:

Undefined behavior, or A bus fault, depending on the architecture (e.g., ARM, RISC-V).

Summary

Aspect	Description
volatile keyword	Prevents compiler from optimizing out memory access.
Pointer type	<code>volatile uint32_t*</code> ensures 32-bit aligned access.
Register access	Direct memory-mapped I/O to control hardware.
Alignment check	Address must be a multiple of 4 for <code>uint32_t</code> .

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

Ans:

```
.text section starting at address 0x00000000
.data section starting at address 0x10000000
```

Minimal Linker Script (linker.ld)

```
SECTIONS
{
    /* Place the .text section at address 0x00000000 */
    .text 0x00000000 :
    {
        *(.text*)      /* Include all .text sections from input files */
    }

    /* Place the .data section at address 0x10000000 */
    .data 0x10000000 :
    {
        *(.data*)      /* Include all .data sections from input files */
    }
}
```

Why Different Addresses for .text and .data?

.text section:

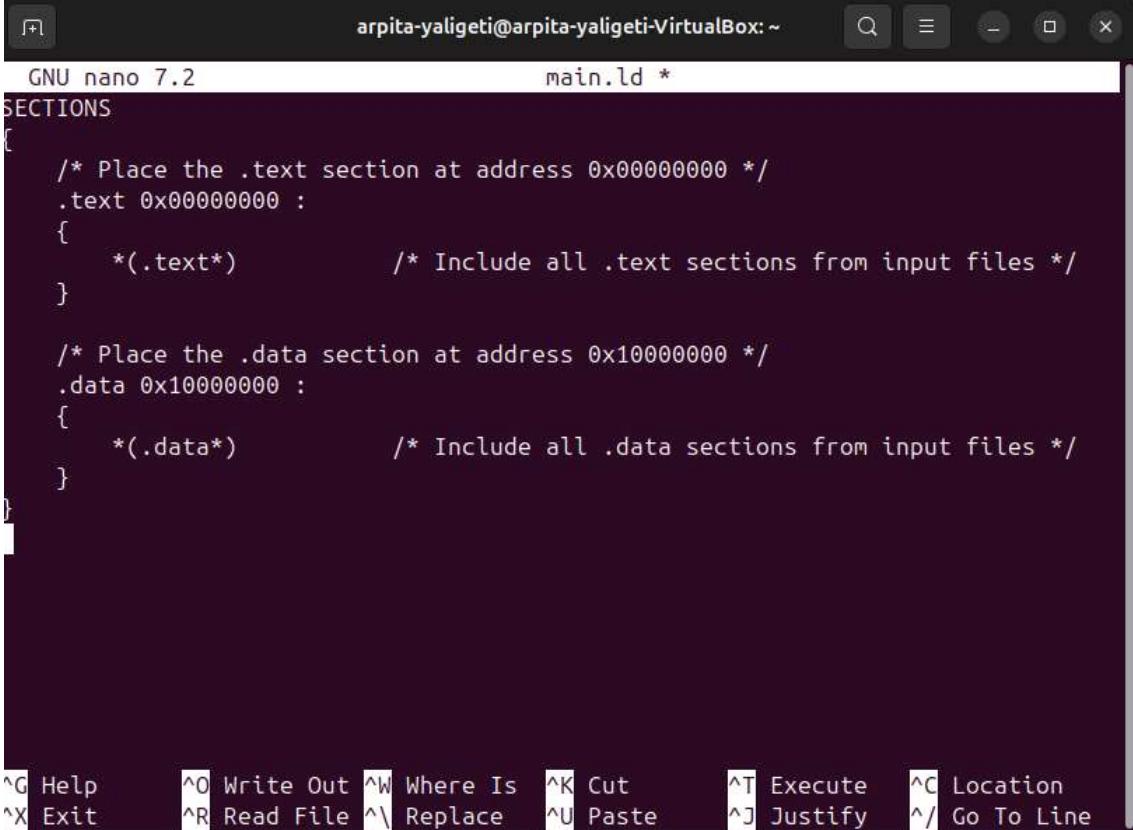
This usually contains program code (instructions). It is typically stored in Flash memory or ROM, which is non-volatile and at a fixed address (here 0x00000000).

.data section:

This contains initialized global/static variables. Since variables must be writable, .data resides in SRAM (RAM), which is volatile memory allowing read-write operations. This is typically mapped at a different higher address (0x10000000 here).

Summary

Section	Purpose	Memory Type	Typical Address Range
.text	Program instructions	Flash (non-volatile)	0x00000000 (example)
.data	Initialized writable data	SRAM (volatile)	0x10000000 (example)



```
GNU nano 7.2                               main.ld *
SECTIONS
{
    /* Place the .text section at address 0x00000000 */
    .text 0x00000000 :
    {
        *(.text*)
        /* Include all .text sections from input files */
    }

    /* Place the .data section at address 0x10000000 */
    .data 0x10000000 :
    {
        *(.data*)
        /* Include all .data sections from input files */
    }
}
```

^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location
 ^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line

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

Ans:

What Does crt0.S Do in a Bare-Metal RISC-V Program?

crt0.S (short for "C RunTime Zero") is a startup assembly file that runs before main() in a bare-metal environment (i.e., no OS). It's essential for setting up the environment for C code execution.

Key Responsibilities of crt0.S

Set up the stack pointer (sp): Needed because the C code relies on the stack (e.g., for function calls, local variables).

Usually set to the top of SRAM.

Zero out the .bss section: bss contains uninitialized global/static variables, which must be zeroed according to the C standard.

Copy .data from Flash to RAM (optional): If .data was initialized in Flash, it must be copied to SRAM for run-time use.

Call main(): The real entry point of the application.

Infinite loop (or wfi) after main() exits: In bare-metal systems, you typically don't return from main().

Example crt0.S (simplified for RV32IMC)

What to Write and Where

crt0.S — Startup file (in assembly)

Set up stack, zero .bss, call main

Save as crt0.S (capital S for preprocessed assembly)

```
.section .text
.globl _start
_start:
    la sp, _stack_top
```

```
    la a0, _bss_start
    la a1, _bss_end
    li a2, 0
```

```
bss_clear:
    bge a0, a1, bss_done
    sw a2, 0(a0)
    addi a0, a0, 4
    j bss_clear
```

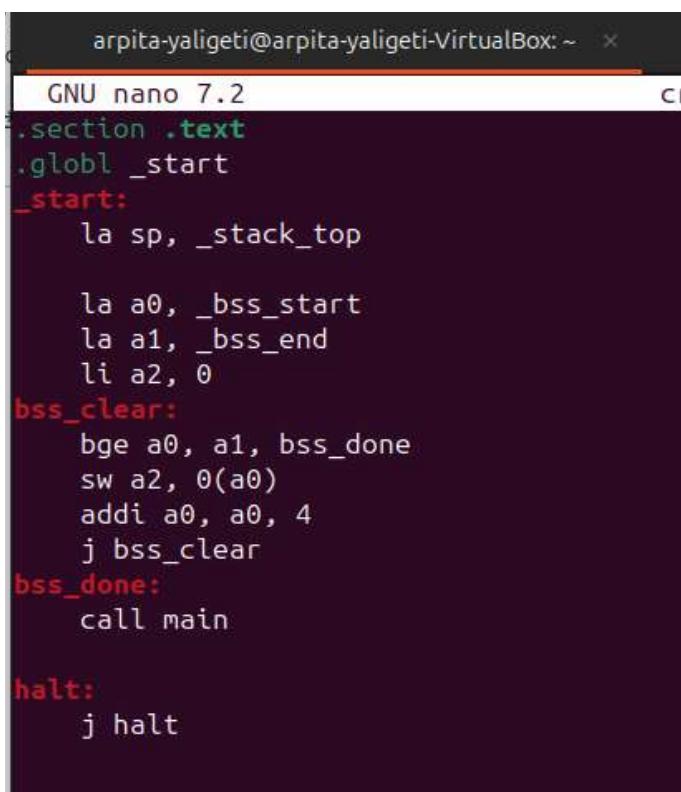
```
bss_done:
    call main
```

halt:

```
    j halt
```

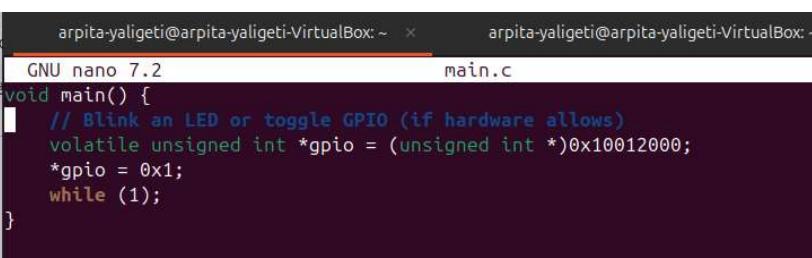
main.c — Your C program

```
void main() {
```



The terminal window shows the assembly code for crt0.S. The code starts with a section directive for .text, followed by a global symbol _start. The _start label is defined with a stack setup (la sp, _stack_top), loading addresses for a0, a1, and a2, and a bss_clear label. The bss_clear label contains a loop that compares a0 and a1, writes to memory at address a2, adds 4 to a0, and loops back until a0 is greater than or equal to a1. After the bss_clear loop, there is a bss_done label with a call to main. Finally, there is a halt label with a jump to halt. The assembly code uses standard RISC-V instructions like la, sw, addi, and j.

```
arpita-yaligeti@arpita-yaligeti-VirtualBox: ~
GNU nano 7.2
.section .text
.globl _start
_start:
    la sp, _stack_top
    la a0, _bss_start
    la a1, _bss_end
    li a2, 0
bss_clear:
    bge a0, a1, bss_done
    sw a2, 0(a0)
    addi a0, a0, 4
    j bss_clear
bss_done:
    call main
halt:
    j halt
```



The terminal window shows the C code for main.c. It defines a main function that includes a comment about toggling GPIO. Inside the main function, it declares a volatile unsigned int *gpio pointing to 0x10012000, sets *gpio to 0x1, and enters a while(1) loop.

```
arpita-yaligeti@arpita-yaligeti-VirtualBox: ~          arpita-yaligeti@arpita-yaligeti-VirtualBox: ~
GNU nano 7.2                                         main.c
void main() {
    // Blink an LED or toggle GPIO (if hardware allows)
    volatile unsigned int *gpio = (unsigned int *)0x10012000;
    *gpio = 0x1;
}
```

```

// Blink an LED or toggle GPIO (if hardware allows)
volatile unsigned int *gpio = (unsigned int *)0x10012000;
*gpio = 0x1;
while (1);
}

```

main.ld — Linker script

```

SECTIONS {
    . = 0x00000000;
    .text : {
        *(.text*)
    }

    . = 0x10000000;
    .data : {
        *(.data*)
    }

    .bss : {
        _bss_start = .;
        *(.bss*)
        _bss_end = .;
    }

    . = 0x10004000;
    _stack_top = .;
}

```

```

arpita-yaligeti@arpita-yaligeti-VirtualBox: ~ ×
GNU nano 7.2
SECTIONS {
    . = 0x00000000;
    .text : {
        *(.text*)
    }

    . = 0x10000000;
    .data : {
        *(.data*)
    }

    .bss : {
        _bss_start = .;
        *(.bss*)
        _bss_end = .;
    }

    . = 0x10004000;
    _stack_top = .;
}

```

Where to Get crt0.S

From Newlib (embedded C library):

Look in <newlib source>/libgloss/riscv/crt0.S

Newlib provides a portable and well-maintained crt0.S for RISC-V.

From Board/Device SDKs

E.g., SiFive Freedom E SDK, Arduino RISC-V cores, or RISC-V template projects on GitHub.

These often include startup files tailored to the device's memory map and peripherals.

DIY

For learning or minimal cases, write your own based on your linker script and memory map.

Summary

Role of crt0.S

Description

Stack setup	Set initial sp
BSS init	Zero .bss section
Data copy (optional)	Copy .data from Flash to RAM
Entry to main()	Hand over to your C program
Infinite loop/halt	Prevents undefined behavior after main()

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

Ans: To enable and handle **MTIP (Machine Timer Interrupt)**, you must:

- | Step | Action |
|------|--|
| 1 | Set mtimecmp to desired time |
| 2 | Enable machine-timer interrupt in mie (MIE_MTIE bit) |
| 3 | Enable global interrupts via mstatus |
| 4 | Write an interrupt handler (mtimer_handler) |
| 5 | Register the handler at trap entry (mtvec) |

main.c

```
#include <stdint.h>

// Memory-mapped CLINT
addresses (platform-specific)
#define MTIMECMP ((volatile uint64_t *)0x02004000)
#define MTIME      ((volatile uint64_t *)0x0200BFF8)
#define MSTATUS_MIE (1 << 3)
#define MIE_MTIE   (1 << 7)

// External trap handler in
assembly
extern void trap_entry(void);

// Simple handler in C
void __attribute__((interrupt)) mtimer_handler(void) {
    volatile uint32_t *gpio = (uint32_t *)0x10012000;
    *gpio ^= 0x1; // Toggle GPIO for visibility (e.g., LED)

    // Set next timer interrupt
    *MTIMECMP = *MTIME + 1000000; // Simple periodic interval
}

void main() {
```

```
// Set next timer interrupt
*MTIMECMP = *MTIME + 1000000; // Simple periodic interval
}

void main() {
    // Set timer: next interrupt after 1 million ticks
    *MTIMECMP = *MTIME + 1000000;
```

```

volatile uint32_t *gpio = (uint32_t *)0x10012000;
*gpio ^= 0x1; // Toggle GPIO for visibility (e.g., LED)

// Set next timer interrupt
*MTIMECMP = *MTIME + 1000000; // Simple periodic interval
}

void main() {
    // Set timer: next interrupt after 1 million ticks
    *MTIMECMP = *MTIME + 1000000;

    // Enable timer interrupt
    asm volatile("csrs mie, %0" :: "r"(MIE_MTIE));

    // Enable global machine interrupts
    asm volatile("csrs mstatus, %0" :: "r"(MSTATUS_MIE));

    // Set the machine trap-vector base address
    asm volatile("la t0, trap_entry\ncsrw mtvec, t0");

    while (1); // Loop forever, wait for interrupts
}

```

trap.S — Trap Entry in Assembly

```

.section .text
.globl trap_entry

trap_entry:
    csrr t0, mcause
    li t1, 0x80000007      # Machine timer interrupt (bit 31 set, cause = 7)
    bne t0, t1, other_trap

    call mtimer_handler     # Call C timer interrupt handler

    mret                   # Return from trap

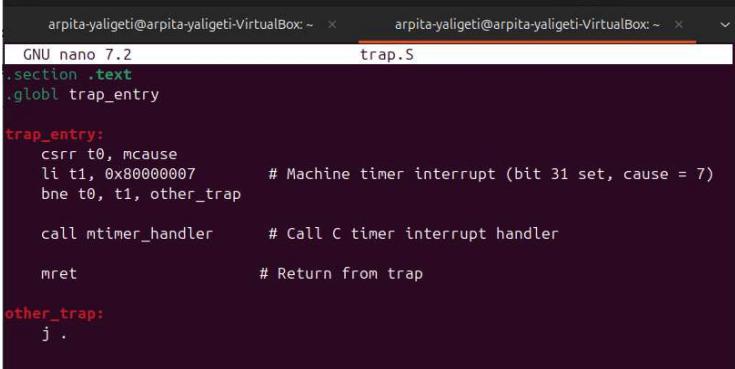
other_trap:
    j .

    call mtimer_handler   # Call
C timer interrupt handler

    mret                   # Return from trap

other_trap:
    j .

```



Element	Description
---------	-------------

MTIMEC If mtime >= mtimetcmp, an interrupt is raised
MP

mie.MT Bit 7 enables timer interrupts
IE

mstatus.M Global interrupt enable
IE

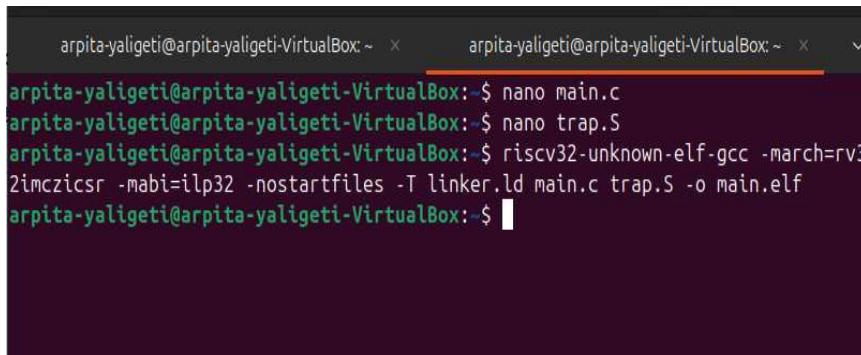
mcause = Machine-timer interrupt
0x80000007

`__attribute__((interrupt))` Tells GCC this is an ISR with correct prologue/epilogue

mtve Trap-vector base address register
c

Build with:

```
riscv32-unknown-elf-gcc -march=rv32imczicsr -mabi=ilp32 -nostartfiles -T linker.ld main.c  
trap.S -o main.elf
```



The terminal window shows the following command sequence:

```
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano main.c  
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano trap.S  
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ riscv32-unknown-elf-gcc -march=rv32imczicsr -mabi=ilp32 -nostartfiles -T linker.ld main.c trap.S -o main.elf  
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$
```



No "output" on screen by default
Bare-metal code usually doesn't have a console or screen unless you add UART or similar I/O.
So no text output appears unless you implement serial communication.

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. Provide a two-thread mutex example (pseudo-threads in main) using lr/sc on RV32
Ans. two-thread mutex example for RV32 using a spin-lock implemented with lr.w / sc.w in C.
The threads are simulated by two functions called from main().

Spin-lock Based Mutex (using lr.w / sc.w)

```
#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 lock
            "sc.w %[val], %[val], (%[addr])\n" // Store-cond if still reserved
            "1:"                               

            : [val] "=r" (tmp)
            : [addr] "r" (lock_addr)
            : "memory"
        );

        } while (tmp != 0); // Retry if store failed (lock taken)

    }

    // Unlock (just clear lock variable)

    void unlock_mutex(volatile int *lock_addr) {

        *lock_addr = 0;
    }

    // Pseudo-thread A

    void thread_A() {

        lock_mutex(&lock);

        shared_counter += 1; // Critical section
    }
}

```

```

unlock_mutex(&lock);

}

// Pseudo-thread B

void thread_B() {

    lock_mutex(&lock);

    shared_counter += 2; // Critical section

    unlock_mutex(&lock);

}

int main() {

    thread_A(); // Simulate first thread

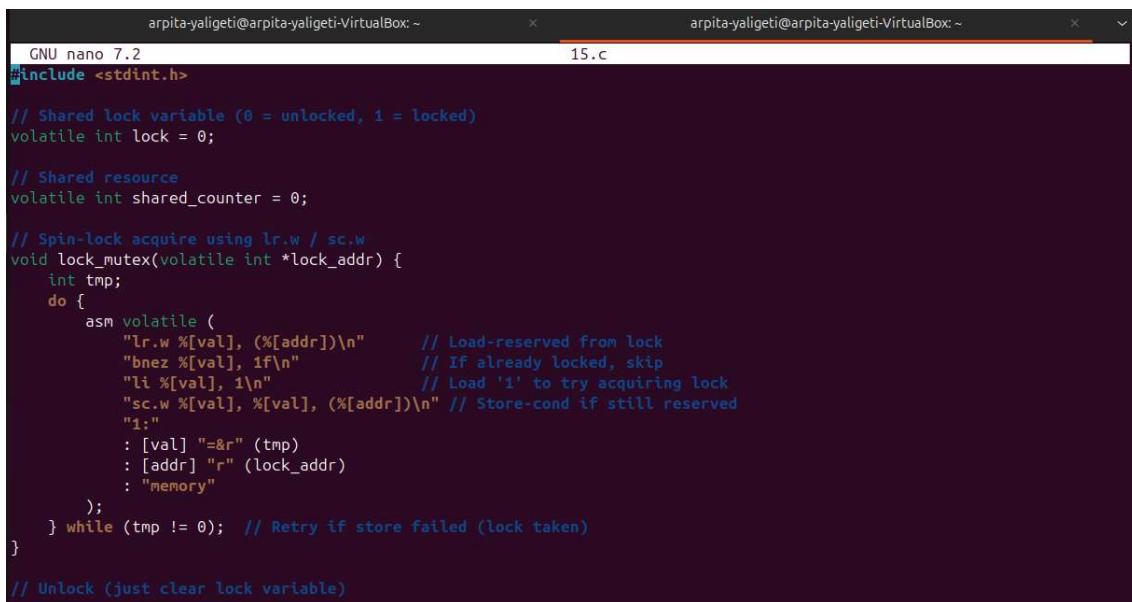
    thread_B(); // Simulate second thread

    // At this point, shared_counter should be 3

    while (1);

}

```



The screenshot shows a terminal window with two tabs. The left tab is titled 'GNU nano 7.2' and contains C code for a spin-lock. The right tab is titled '15.c' and displays the generated assembly code. The assembly code is annotated with comments explaining the assembly instructions:

```

#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
            "bneq %[val], if\n"              // If already locked, skip
            "li %[val], 1\n"                // Load '1' to try acquiring lock
            "sc.w %[val], %[val], (%[addr])\n" // Store-cond if still reserved
            "1:\n"
            : [val] "=r" (tmp)
            : [addr] "r" (lock_addr)
            : "memory"
        );
    } while (tmp != 0); // Retry if store failed (lock taken)
}

// Unlock (just clear lock variable)

```

```

GNU nano 7.2                               15.c
// Unlock (just clear lock variable)
void unlock_mutex(volatile int *lock_addr) {
    *lock_addr = 0;
}

// Pseudo-thread A
void thread_A() {
    lock_mutex(&lock);
    shared_counter += 1; // Critical section
    unlock_mutex(&lock);
}

// Pseudo-thread B
void thread_B() {
    lock_mutex(&lock);
    shared_counter += 2; // Critical section
    unlock_mutex(&lock);
}

int main() {
    thread_A(); // Simulate first thread
    thread_B(); // Simulate second thread

    // At this point, shared_counter should be 3
    while (1);
}

```

Explanation: 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

16. How do I retarget _write so that printf sends bytes to my memory-mapped UART?"

Ans.

Practical Summary: Retargeting `__write()` for `printf` to UART

1. UART Register Mapping

C

```

#define UART_TX_ADDR    0x10000000
#define UART_STATUS_ADDR 0x10000005
#define UART_TX_READY_BIT (1 << 5)

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

```

2. Custom `__write` Implementation

```
c

#include <unistd.h>

int __write(int file, const char *ptr, int len) {

    if (file != 1 && file != 2) // stdout or stderr
        return -1;

    for (int i = 0; i < len; i++) {
        // Wait until UART is ready to transmit
        while (!(uart_status_reg & UART_TX_READY_BIT));
        // Send the character
        *uart_tx_reg = ptr[i];
    }
    return len;
}
```

Integration with Toolchain

3. Linker Command

Ensure you're using:

```
bash
riscv32-unknown-elf-gcc -march=rv32ima -mabi=ilp32 \
-nostartfiles -T linker.ld \
crt0.S main.c syscalls.c -o prog.elf -lc -lgcc
```

`-nostartfiles` → bypass standard startup (you provide `crt0.S`)

`-lc` and `-lgcc` → link `newlib` and GCC support library

Your `__write` overrides `newlib`'s weak version

4. Test Main Function

c

```
#include <stdio.h>
```

```
int main() {
    printf("Hello, UART!\n");
    while (1);
}
```

You must also implement `__exit`, `__sbrk`, etc., to avoid linker errors from `newlib`.

Test on QEMU (`qemu-system-riscv32 -nographic`) if you're simulating UART.

If UART is FIFO-based, consider adding delay or FIFO-full checking.

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

Ans.

1. Use nano or any editor:

```
nano endian.c
```

2. Paste this code:

```
#include <stdio.h>
```

```
#include <stdint.h>
```

```
int main() {
```

```
    union {
```

```
        uint32_t value;
```

The screenshot shows a terminal window titled 'arpita-yaligeti@arpita-yaligeti-VirtualBox: ~'. The window contains the code for 'endian.c'. The code includes #include <stdio.h> and #include <stdint.h>. It defines a union 'test' containing a uint32_t 'value' and a uint8_t array 'bytes[4]'. The 'value' is initialized to 0x01020304. A printf statement outputs the byte order: %02x %02x %02x %02x\n". The terminal window has a dark theme with white text and a light gray background.

```
GNU nano 7.2
#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",
```

```

        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;
}

```

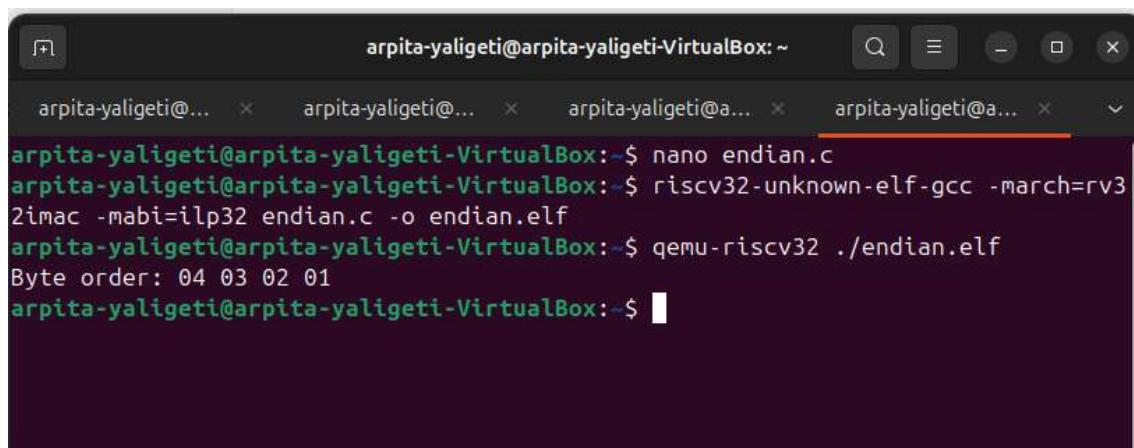
3. Compile

riscv32-unknown-elf-gcc -march=rv32imac -mabi=ilp32 endian.c -o endian.elf

Don't use -nostartfiles here because we want to run it as a Linux-style program.

4. Run with QEMU

qemu-riscv32 ./endian.elf



```

arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ nano endian.c
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ riscv32-unknown-elf-gcc -march=rv32imac -mabi=ilp32 endian.c -o endian.elf
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ qemu-riscv32 ./endian.elf
Byte order: 04 03 02 01
arpita-yaligeti@arpita-yaligeti-VirtualBox:~$ 

```

Summary

Method	Output Visible Where?	Needs Syscalls?	Good For
<code>qemu-riscv32</code>	Terminal stdout	No	Linux-style apps
<code>qemu-system-riscv32</code>	Terminal via -nographic + UART	Yes (<code>_write</code>)	Bare-metal, embedded code
<code>-d int -D qemu.log</code>	In <code>qemu.log</code> file	No	Debugging