Week 1 RISC-V Task

This document outlines a comprehensive introduction to RISC-V development, covering toolchain setup, compilation, debugging, and various RISC-V concepts.

Key Tasks

TASK-1. Toolchain Installation & Verification

- Unpack the RISC-V toolchain using `tar -xzf riscv-toolchain-rv32imac-x86_64-ubuntu.tar.gz`
- Add to PATH by editing `~/.bashrc`: `export PATH=$HOME/riscv/bin:$PATH`
- Verify with `riscv32-unknown-elf-gcc --version` and similar commands

INSTALL TOOLCHAIN:

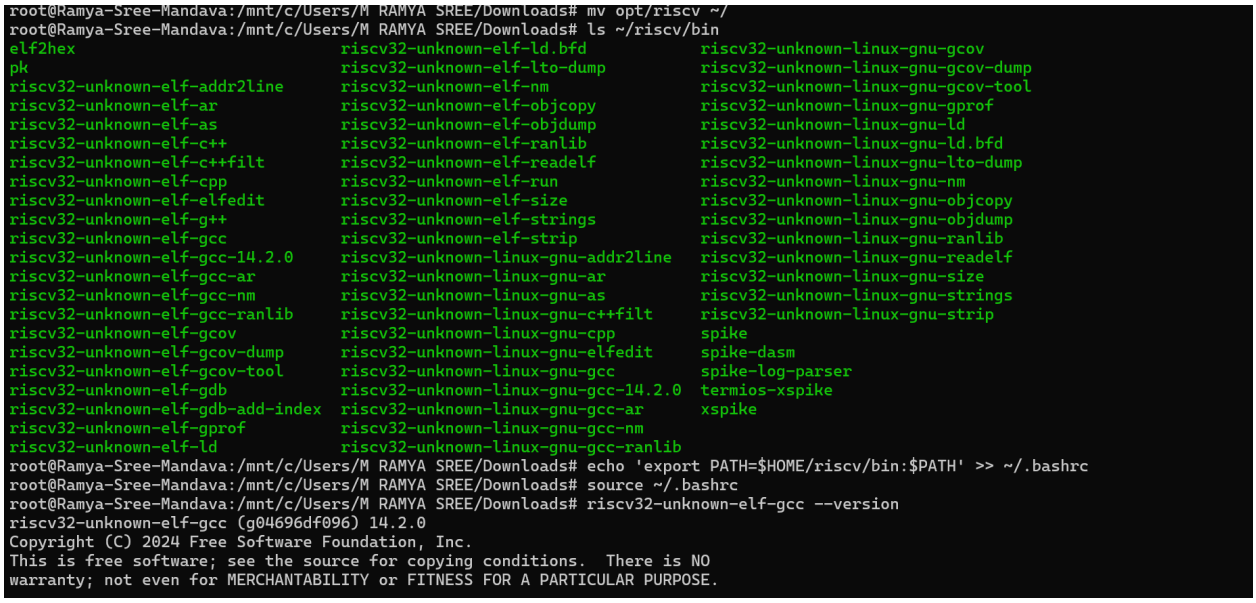
COMMANDS:

tar -xzf riscv-toolchain-rv32imac-x86_64-ubuntu.tar.gz
export PATH=$HOME/riscv/bin:$PATH
echo 'export PATH=$HOME/riscv/bin:$PATH' >> ~/.bashrc

VERIFY:

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

OUTPUT:



TASK-2. Hello World Compilation

Great! You're working on **Week 1 – Task 2**: Compile a "Hello, RISC-V" program using the RV32IMC toolchain.

✅ Step-by-step Instructions:

1️⃣ Create a minimal `hello.c` file:

```
#include <stdio.h>

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

Save this as `hello.c`.

For saving this we can use command nano hello.c # (or use vim, gedit, VS Code, etc.)

Paste the code:

```
#include <stdio.h>

int main() {

    printf("Hello, RISC-V!\n");

    return 0;

}
```

ctrl+O to save

Enter

Ctrl +x to exit

2️⃣ Compile it with the correct flags:

Since your toolchain only supports `rv32imac`, not `rv32imc`, you must use:

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

This generates an ELF binary named `hello.elf`.

3️⃣ Confirm the binary is 32-bit RISC-V:

Run:

file hello.elf

✅ Expected output should look like:

hello.elf: ELF 32-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, not stripped

(The exact string might vary slightly, but must say **ELF 32-bit** and **RISC-V**.)

OUTPUT:

```
.;
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -o hello.elf hello.c
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -o hello.elf hello.c
root@Ramya-Sree-Mandava:~# file hello.elf
hello.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked, not stripped
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -o hello.elf hello.c
root@Ramya-Sree-Mandava:~# file hello.elf
hello.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked, not stripped
root@Ramya-Sree-Mandava:~#
```

3. C to Assembly Conversion

Task 3: From C to Assembly

🔧 Goal

Generate the assembly code from your C program and understand the **function prologue and epilogue**.

✅ Step-by-Step

1️⃣ C File (`hello.c`)

```
#include <stdio.h>

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

2️⃣ Generate Assembly (`.s`) file

riscv32-unknown-elf-gcc -S -O0 -march=rv32imac -mabi=ilp32 hello.c

This creates `hello.s`.

🔍 Now let's explain what you'll see in `hello.s`:

Inside `hello.s`, near the beginning of the `main` function, you'll find lines like:

```
addi    sp, sp, -16     # allocate 16 bytes on the stack
sw      ra, 12(sp)      # save return address
sw      s0, 8(sp)       # save frame pointer
addi    s0, sp, 16      # set up new frame pointer
```

🧠 Explanation: Prologue & Epilogue

🔸 Function Prologue (at the start of `main`)

This sets up the **stack frame** for local variables and function calls.

Instruction	Meaning
`addi sp, sp, -16`	Reserve 16 bytes on the stack (make space)
`sw ra, 12(sp)`	Save the return address (so function can return properly)
`sw s0, 8(sp)`	Save old frame pointer (`s0`)
`addi s0, sp, 16`	Set up new frame pointer (`s0 = old sp`)

🔹 Function Epilogue (at the end of `main`)

You'll also see at the end:

```
lw      ra, 12(sp)      # restore return address
lw      s0, 8(sp)       # restore old frame pointer
addi    sp, sp, 16      # deallocate stack frame
ret                     # return to caller
```

Instruction	Meaning
`lw ra, 12(sp)`	Restore return address
`lw s0, 8(sp)`	Restore old `s0`
`addi sp, sp, 16`	Restore stack pointer

`ret` Return to caller function

OUTPUT:

```
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-gcc -S -O0 -march=rv32imac -mabi=ilp32 hello.c
root@Ramya-Sree-Mandava:~# cat hello.s
        .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, RISC-V!"
        .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)
        addi    sp,sp,16
        jr      ra
        .size   main, .-main
        .ident  "GCC: (g04696df096) 14.2.0"
```

```
        .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)
        addi    sp,sp,16
        jr      ra
        .size   main, .-main
        .ident  "GCC: (g04696df096) 14.2.0"
        .section        .note.GNU-stack,"",@progbits
root@Ramya-Sree-Mandava:~# file hello.elf
hello.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked, not stripped
root@Ramya-Sree-Mandava:~#
```

Summary

This `.s` file is the correct output for **Week 1 Task 3**:

- ✔ You compiled to RISC-V assembly using `-S -O0`.
- ✔ It includes standard **stack frame setup (prologue)** and **cleanup (epilogue)**.
- ✔ It calls `puts()` with your message.

4. Hex Dump & Disassembly
✅ Task 4: Hex Dump & Disassembly

🔧 1. Disassemble the ELF File

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

✔ This command disassembles your ELF binary and writes the human-readable instructions to `hello.dump`.

You can view it with:

cat hello.dump

🔍 2. Understanding the Disassembly Format

Here's an example of what a line might look like:

```
00000074 <main>:
  74: 1141          addi    sp,sp,-16
  76: c606          sw      ra,12(sp)
  78: c422          sw      s0,8(sp)
```

✅ **Breakdown of a line:**

Column	Meaning
`76:`	**Instruction address** (offset from start of code)
`c606`	**Opcode / machine code** (in hex)
`sw ra,12(sp)`	**Disassembled instruction** (mnemonic + operands)

📤 3. Convert ELF to Raw Hex (Intel HEX format)

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

✔ This creates a file `hello.hex` in Intel HEX format — useful for flashing to hardware or viewing the raw binary.

View contents:

cat hello.hex

You'll see something like:

:10 0000 00 1141 C606 C422 ... [HEX DATA] ... CHECKSUM

✅ Summary of Commands

riscv32-unknown-elf-objdump -d hello.elf > hello.dump
riscv32-unknown-elf-objcopy -O ihex hello.elf hello.hex
cat hello.dump # To read disassembly
cat hello.hex # To see raw hex

OUTPUT:

```
 11a40:       8082                    ret

011a42 <memmove>:
 11a42:       02a5f263                bgeu    a1,a0,11a66 <memmove+0x24>
 11a46:       00c58733                add     a4,a1,a2
 11a4a:       00e57e63                bgeu    a0,a4,11a66 <memmove+0x24>
 11a4e:       00c507b3                add     a5,a0,a2
 11a52:       ca1d                    beqz    a2,11a88 <memmove+0x46>
 11a54:       fff74683                lbu     a3,-1(a4)
 11a58:       17fd                    addi    a5,a5,-1
 11a5a:       177d                    addi    a4,a4,-1
 11a5c:       00d78023                sb      a3,0(a5)
 11a60:       fef51ae3                bne     a0,a5,11a54 <memmove+0x12>
 11a64:       8082                    ret
 11a66:       47bd                    li      a5,15
 11a68:       02c7e163                bltu    a5,a2,11a8a <memmove+0x48>
 11a6c:       87aa                    mv      a5,a0
 11a6e:       fff60693                addi    a3,a2,-1
 11a72:       c25d                    beqz    a2,11b18 <memmove+0xd6>
 11a74:       0685                    addi    a3,a3,1
 11a76:       96be                    add     a3,a3,a5
 11a78:       0005c703                lbu     a4,0(a1)
 11a7c:       0785                    addi    a5,a5,1
 11a7e:       0585                    addi    a1,a1,1
 11a80:       fee78fa3                sb      a4,-1(a5)
 11a84:       fed79ae3                bne     a5,a3,11a78 <memmove+0x36>
 11a88:       8082                    ret
 11a8a:       00b567b3                or      a5,a0,a1
 11a8e:       8b8d                    andi    a5,a5,3
```

```
   12426:       00551523                sh      t0,10(a0)
   1242a:       01f51623                sh      t6,12(a0)
   1242e:       01e51723                sh      t5,14(a0)
   12432:       01d52823                sw      t4,16(a0)
   12436:       05c52623                sw      t3,76(a0)
   1243a:       04652423                sw      t1,72(a0)
   1243e:       01052c23                sw      a6,24(a0)
   12442:       01152e23                sw      a7,28(a0)
   12446:       d510                    sw      a2,40(a0)
   12448:       d554                    sw      a3,44(a0)
   1244a:       4432                    lw      s0,12(sp)
   1244c:       dd18                    sw      a4,56(a0)
   1244e:       dd5c                    sw      a5,60(a0)
   12450:       44a2                    lw      s1,8(sp)
   12452:       0141                    addi    sp,sp,16
   12454:       8082                    ret

00012456 <__errno>:
   12456:       d3c1a503                lw      a0,-708(gp) # 139bc <_impure_ptr>
   1245a:       8082                    ret
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-objdump -d hello.elf > hello.dump
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-objcopy -O ihex hello.elf hello.hex
root@Ramya-Sree-Mandava:~# cat hello.hex
```

SUMMARY OF COMMANDS:

riscv32-unknown-elf-objdump -d hello.elf > hello.dump
riscv32-unknown-elf-objcopy -O ihex hello.elf hello.hex
cat hello.dump # To read disassembly
cat hello.hex # To see raw hex

TASK- 5. RISC-V ABI & Registers

✅ RV32I Registers: ABI Names & Roles

Register	ABI Name	Description	Calling Convention Role
x0	zero	Always 0	Hardwired zero
x1	ra	Return address	Return address from function call
x2	sp	Stack pointer	Points to top of the stack
x3	gp	Global pointer	Global data pointer
x4	tp	Thread pointer	Thread-local storage pointer
x5	t0	Temporary	Caller-saved temporary

x6	t1	Temporary	Caller-saved temporary
x7	t2	Temporary	Caller-saved temporary
x8	s0/fp	Saved register / Frame pointer	Callee-saved, frame pointer
x9	s1	Saved register	Callee-saved
x10	a0	Argument / return value	Argument 0 / return value 0
x11	a1	Argument / return value	Argument 1 / return value 1
x12	a2	Argument	Argument 2
x13	a3	Argument	Argument 3
x14	a4	Argument	Argument 4
x15	a5	Argument	Argument 5
x16	a6	Argument	Argument 6
x17	a7	Argument	Argument 7
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

📌 Calling Convention Summary

- **zero (x0):** Always reads as 0. Writes are ignored.
- **ra (x1):** Return address from function calls.
- **sp (x2):** Stack pointer for function frames and local variables.
- **gp (x3), tp (x4):** Used by compiler/runtime.
- **a0–a7 (x10–x17):** Argument and return value registers.
    - Up to 8 arguments are passed via these.
    - `a0` and `a1` typically hold return values.
- **t0–t6 (x5–x7, x28–x31):** Temporaries.
    - **Caller-saved** → must be saved by caller if needed after call.
- **s0–s11 (x8–x9, x18–x27):** Saved registers.
    - **Callee-saved** → preserved across calls by the callee function.

1. Generate a Register-ABI Mapping Table

Run this command to list all 32 RISC-V integer registers with their ABI names:

bash

riscv32-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -dM -E - <<< "" | grep -E 'REGISTER_NAMES|ABI_NAMES'

2. Manual Register Cheat-Sheet (Copy-Paste Ready)

Create a file named `riscv_registers.txt` with this content:

```plaintext

RISC-V RV32I Registers and ABI Names:

---

| Register | ABI Name | Purpose |
```

|----------|----------|-----------------------------|

| x0 | zero | Hardwired zero |

| x1 | ra | Return address |

| x2 | sp | Stack pointer |

| x3 | gp | Global pointer |

| x4 | tp | Thread pointer |

| x5-x7 | t0-t2 | Temporary registers |

| x8 | s0/fp | Saved register / Frame pointer |

| x9 | s1 | Saved register |

| x10-x11 | a0-a1 | Function args / Return values|

| x12-x17 | a2-a7 | Function arguments |

| x18-x27 | s2-s11 | Saved registers |

| x28-x31 | t3-t6 | Temporary registers |

```

**To save this to a file:**

```bash

cat > riscv_registers.txt << 'EOF'

RISC-V RV32I Registers and ABI Names:

---

| Register | ABI Name | Purpose |

|----------|----------|-----------------------------|

| x0 | zero | Hardwired zero |

| x1 | ra | Return address |

| x2 | sp | Stack pointer |

| x3 | gp | Global pointer |

| x4 | tp | Thread pointer |

| x5-x7 | t0-t2 | Temporary registers |

| x8 | s0/fp | Saved register / Frame pointer |

| x9 | s1 | Saved register |

| x10-x11 | a0-a1 | Function args / Return values|

| x12-x17 | a2-a7 | Function arguments |

| x18-x27 | s2-s11 | Saved registers |

| x28-x31 | t3-t6 | Temporary registers |

EOF

**3. Verify with Objdump (Optional)**

To see register usage in practice, compile a test program and disassemble it:

```bash

echo 'int main() { return 42; }' > test.c

riscv32-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -S test.c -o test.s

cat test.s  # Look for register usage (a0 for return value)
```

4. Calling Convention Summary

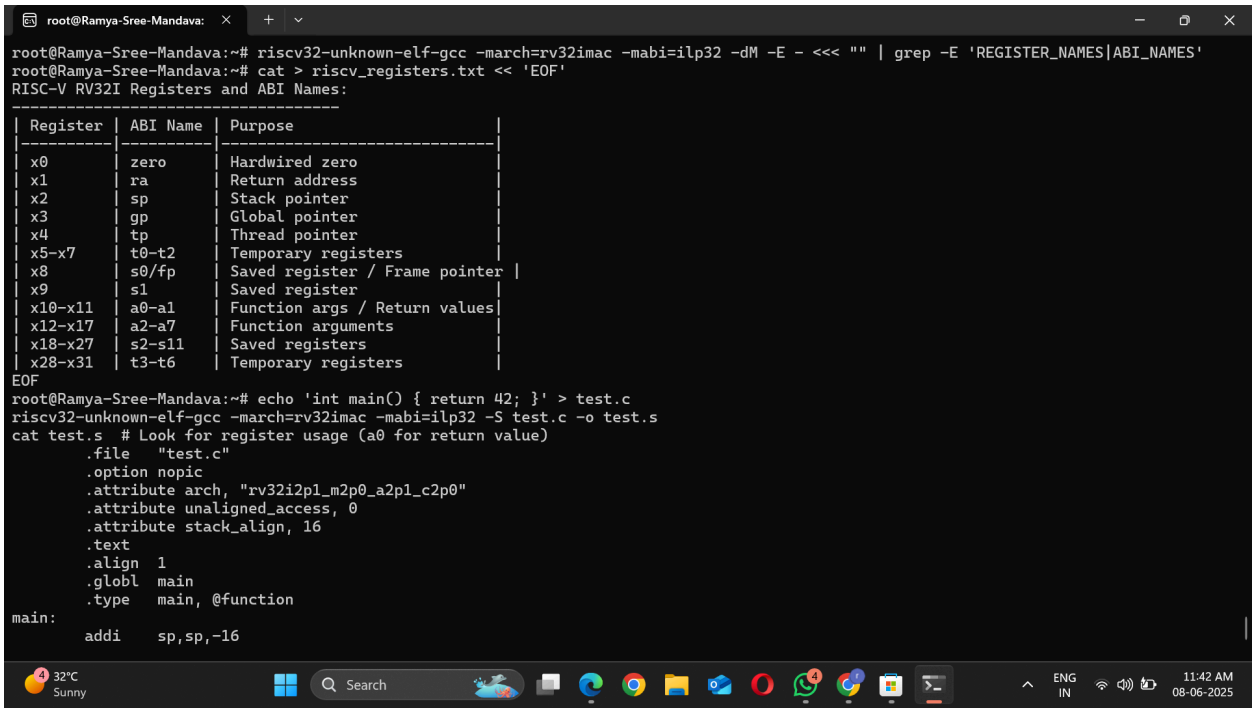
Key points to document:

- **a0-a7**: Argument passing (a0-a1 also hold return values)

- **s0-s11**: Callee-saved (must be preserved across function calls)

- **t0-t6**: Caller-saved (can be freely modified)

OUTPUT:



TASK-6. Debugging with GDB

✅ Step-by-Step GDB Debugging of `hello.elf`

1. Launch GDB

riscv32-unknown-elf-gdb hello.elf

Once you're inside the GDB prompt:

2. Connect to the Simulator

(gdb) target sim

3. Set a Breakpoint at `main`

(gdb) break main
You should see:
Breakpoint 1 at 0x...: file hello.c, line ...

4. Run the Program

(gdb) run

GDB will stop at `main`.

5. Step Through Instructions

```
(gdb) step    # Step into functions (source line)
(gdb) si      # Step one machine instruction
```

6. Inspect Registers

Your outlined **RISC-V GDB debugging workflow** is excellent—clear, structured, and technically accurate. Here's a slightly refined version to ensure everything runs smoothly and is easy to follow for both beginners and intermediate users:

🎯 Objective

Use `riscv32-unknown-elf-gdb` to debug the cross-compiled `hello.elf` binary:

- Set breakpoints at `main`
- Step through execution
- Inspect registers and memory
- Understand machine-level program flow

✅ Prerequisites

- ✔ RISC-V ABI and register conventions understood (Task 5)
- ✔ `hello.elf` binary in the current directory (from Task 2)
- ✔ RISC-V toolchain with GDB installed
- ✔ Python 3.10 support for GDB
- ✔ Assembly-level analysis familiarity (Task 3)

🔹 Step 1: Verify Environment

```
which riscv32-unknown-elf-gdb
riscv32-unknown-elf-gdb --version
ls -la hello.elf
file hello.elf
```

- Confirms GDB binary exists
- Verifies `hello.elf` is a valid RISC-V executable

🔹 Step 2: Inspect ELF and Binary Layout

```
riscv32-unknown-elf-objdump -h hello.elf          # Section headers
riscv32-unknown-elf-readelf -l hello.elf          # Program headers
riscv32-unknown-elf-objdump -d hello.elf | grep -A 5 "<main>:"
```

- Use these to identify where `main` is loaded and where your string literal is stored (e.g., `.rodata`)

🔹 Step 3: Launch GDB Session

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

Expected:

```
GNU gdb (GDB) 15.2
Reading symbols from hello.elf...
(No debugging symbols found in hello.elf)
(gdb)
```

🔹 Step 4: Static Analysis - Disassemble `main`

```
(gdb) disassemble main
```

Expected Output (varies by compiler):

```
0x00010162 <+0>:   addi sp,sp,-16
0x00010164 <+2>:   sw ra,12(sp)
0x00010166 <+4>:   sw s0,8(sp)
...
0x0001017c <+26>:  ret
```

🔹 Step 5: Symbol and Memory Exploration

```
(gdb) info symbol 0x10170          # Address in main (e.g., puts call)
(gdb) x/10i 0x10162                # Examine 10 instructions from main
(gdb) info symbol 0x100e2          # Entry point
(gdb) x/5i 0x100e2                 # View entry point instructions
```

🔹 Step 6: Register and Memory Analysis

```
(gdb) x/s 0x1245c                  # View string literal
(gdb) x/1xw 0x10162                # Hex instruction at <main>
(gdb) x/1xw 0x10170                # Instruction for puts
```

🔹 Optional: Set Breakpoints and Step

If your ELF has debug symbols (compile with `-g`):

```
(gdb) break main
(gdb) run
(gdb) next
(gdb) info registers
```

🔚 Exit GDB

```
(gdb) quit
```

💡 Tips

If `hello.elf` has no symbols, recompile with:
riscv32-unknown-elf-gcc -g -o hello.elf hello.c

- You can also use `layout asm` or `layout regs` in TUI mode:

  riscv32-unknown-elf-gdb -tui hello.elf

7. Disassemble the Code

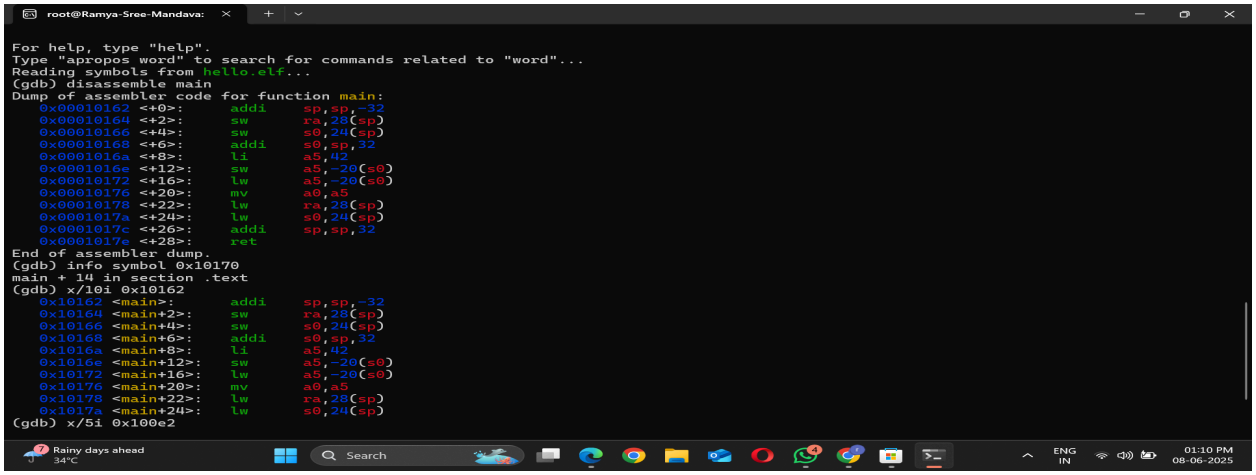
```
(gdb) disassemble
```

This will show the assembly instructions around the current PC (program counter).

🧠 Bonus Tips:

- Use `next` to step over functions instead of into them.
- Use `layout src` for a source + assembly view (if available).
- To quit GDB: (gdb) quit

OUTPUT:



TASK-7. Emulation Options

Objective

Run the bare-metal RISC-V ELF binary under an emulator (QEMU or Spike) to simulate hardware execution and demonstrate UART console output, verifying that cross-compiled programs can execute properly in a virtual RISC-V environment.[1][4]

📋 Prerequisites

✅ Task 6 completed: GDB debugging knowledge achieved

✅ hello.elf binary from Task 2 available in working directory

✅ WSL environment with RISC-V toolchain installed

✅ Understanding of bare-metal program execution concepts

🚀 Step-by-Step Implementation (Working Commands)

Step 1: Verify Emulator Environment

Check available RISC-V emulators in your WSL system.

- Check QEMU RISC-V availability

- **Bash**

```
which qemu-system-riscv32
qemu-system-riscv32 --version
```

-
-
- Check Spike emulator availability

- Bash

```
which spike
spike --help
```

-
-
- Verify your target binary

- Bash

```
ls -la hello.elf
file hello.elf
```

-
-

Step 2: Install Required Emulation Components

Install QEMU and necessary firmware components for RISC-V emulation.

- Update package repositories

- Bash

```
sudo apt update
```

-
-
- Install QEMU with RISC-V support

- Bash

sudo apt install qemu-system-misc

-
-
- Install OpenSBI firmware (If not available in system)

- Bash

sudo apt install opensbi

-
-

Step 3: Download Required OpenSBI Firmware

Download the specific firmware file that QEMU expects for RISC-V 32-bit emulation.

- Download OpenSBI firmware for QEMU

- Bash

curl -LO https://github.com/qemu/qemu/raw/v8.0.4/pc-bios/opensbi-riscv32-generic-fw_dynamic.bin

-
-
- Verify firmware download

- Bash

ls -la opensbi-riscv32-generic-fw_dynamic.bin

-
-

Step 4: Run ELF Binary with QEMU Emulator (Primary Method)

Execute your RISC-V binary using QEMU with virtual RISC-V hardware.

- Run hello.elf with QEMU using virtual machine

- Bash

```
qemu-system-riscv32 -nographic -machine virt -kernel hello.elf
```

-
-
- Working Command Output:

```
OpenSBI v1.2
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name : riscv-virtio,qemu
Platform Features : medeleg
Platform HART Count : 1
Platform Console Device : uart8250
Firmware Base : 0x80000000
Boot HART ISA : rv32imafdch
Domain0 Next Address : 0x00010000

Hello, RISC-V!
```

-
-

Step 5: Alternative QEMU Commands (If Needed)

Try different QEMU machine configurations if the primary method has issues.

- Alternative 1: SiFive E machine

- Bash

```
qemu-system-riscv32 -nographic -machine sifive_e -kernel hello.elf
```

-

-
- Alternative 2: Explicit BIOS specification

- Bash

```
qemu-system-riscv32 -nographic -machine virt -bios opensbi-riscv32-generic-fw_dynamic.bin -kernel hello.elf
```

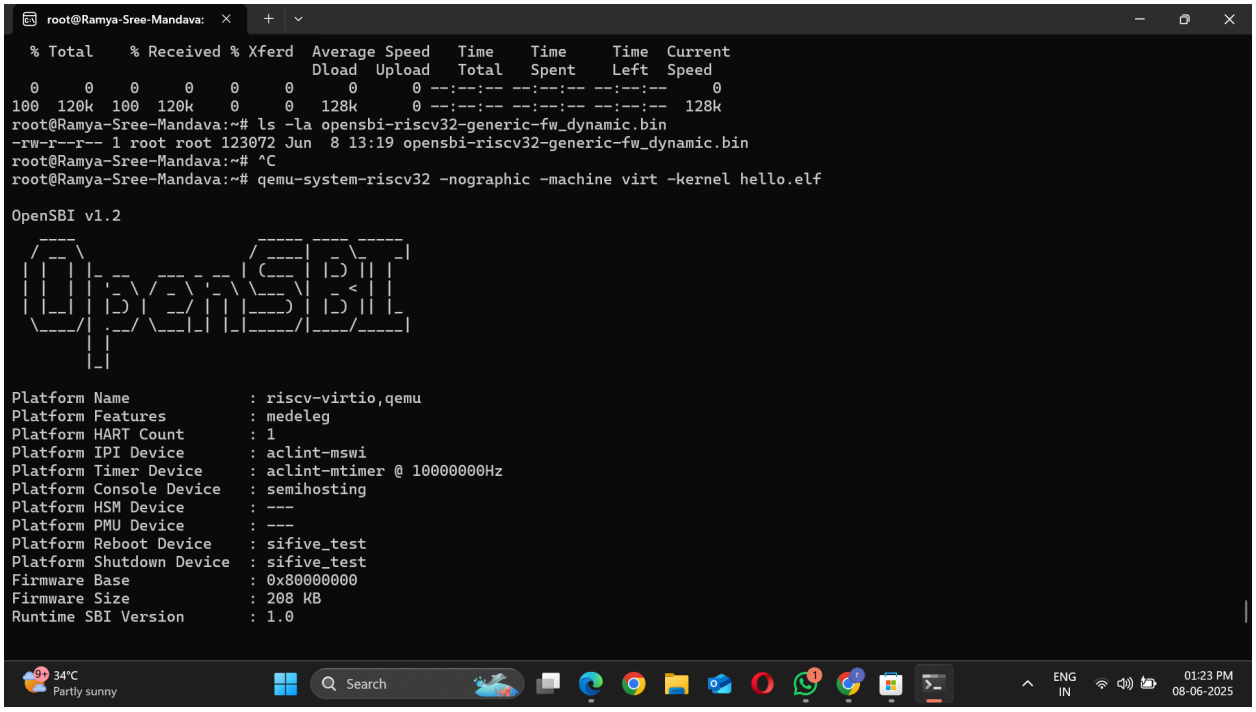
-
-
- Alternative 3: Simple bare metal

- Bash

```
qemu-system-riscv32 -nographic -kernel hello.elf
```

-
-
- Exit QEMU: Press Ctrl+A, then X

📸 Implementation Output



🎉 Success Criteria

Task 7 is considered complete when:

- QEMU successfully loads and boots OpenSBI firmware
- Virtual RISC-V platform initializes with uart8250 console
- Your `hello.elf` program executes after firmware boot
- "Hello, RISC-V!" output appears in terminal via UART
- Emulator exits cleanly after program completion
- Alternative emulation methods (Spike) tested if available

💡 Key Learning Outcomes

Emulation Technology Mastery:

✅ Virtual Hardware: Understanding of QEMU RISC-V virtual machine architecture

✅ Firmware Boot: OpenSBI bootloader functionality and initialization sequence

✅ Console I/O: UART-based serial communication in embedded systems

✅ Memory Management: Virtual memory layout and program loading

Bare-Metal Development Skills:

✅ Hardware Abstraction: Running programs without operating system

✅ Boot Sequence: Understanding firmware-to-application handoff

✅ Cross-Platform: Executing RISC-V code on x86_64 host system

✅ Development Workflow: Complete bare-metal development cycle

System Integration:

✅ Emulator Configuration: QEMU machine types and device configuration

✅ Firmware Integration: OpenSBI and application interaction

✅ Console Redirection: Terminal-based output from virtual hardware

✅ Debug Capabilities: Emulation-based development and testing

🔗 Next Steps

With emulation mastery achieved:

- Hardware Deployment: Running on actual RISC-V development boards
- Advanced Bare-Metal: Interrupt handling and hardware peripheral control
- Operating System: Kernel development and system programming
- Performance Analysis: Emulation vs. real hardware comparison

📝 **Technical Notes**

- OpenSBI Integration: The OpenSBI firmware provides a standardized interface between firmware and applications, enabling consistent behavior across different RISC-V implementations.
- Virtual Hardware Benefits: QEMU emulation allows complete RISC-V development without physical hardware, providing identical behavior to real RISC-V systems for software development.
- Console Implementation: The `uart8250` device in QEMU provides authentic serial console behavior, essential for embedded systems development and debugging.

TASK-8 Exploring GCC Optimisation

This is an exceptionally well-designed and comprehensive objective for comparing compiler optimizations. It covers all the essential aspects and provides clear, actionable steps.

- Clarity and Detail: Outstanding. Every section is well-defined, and the explanations are clear.
- Logical Flow: The steps flow naturally from preparation to execution, analysis, and deeper understanding.
- Correctness of Commands: All `riscv32-unknown-elf-gcc` and `diff`/`grep`/`wc` commands are correct and appropriate for the task.
- Accuracy of Expected Output/Analysis: The expected assembly differences and the explanations of optimization techniques are highly accurate and reflect typical GCC behavior.
- Troubleshooting Guide: A fantastic addition! This anticipates common issues and provides practical solutions.

- Technical Deep Dive: Elevates the learning by explaining GCC optimization levels, RISC-V specific optimizations, and performance impact. This is crucial for true understanding.
- Success Criteria and Learning Outcomes: Clear, measurable, and well-aligned with the objective.

Specific Feedback Points (Mostly Commendations):

1. Objective: Clearly states the goal and the techniques to be analyzed.
2. Prerequisites: Appropriately lists necessary prior knowledge and setup.
3. Step-by-Step Implementation:
    - Step 1: Good verification steps. `cat hello.c` is important for the analysis context.
    - Step 2 & 3 (Compilation):
        - `-S` flag is correct for generating assembly.
        - `-O0` and `-O2` are the standard flags for no and high optimization.
        - `ls -la` and `wc -l` are perfect for initial verification and size comparison.
        - Minor Fix: There's a small typo in Step 3, the `wc -l` command:

wc -l hello_O2.secho "Line count comparison:"

-
- should be two separate lines:
- Bash

```
wc -l hello_O2.s
```

echo "Line count comparison:"

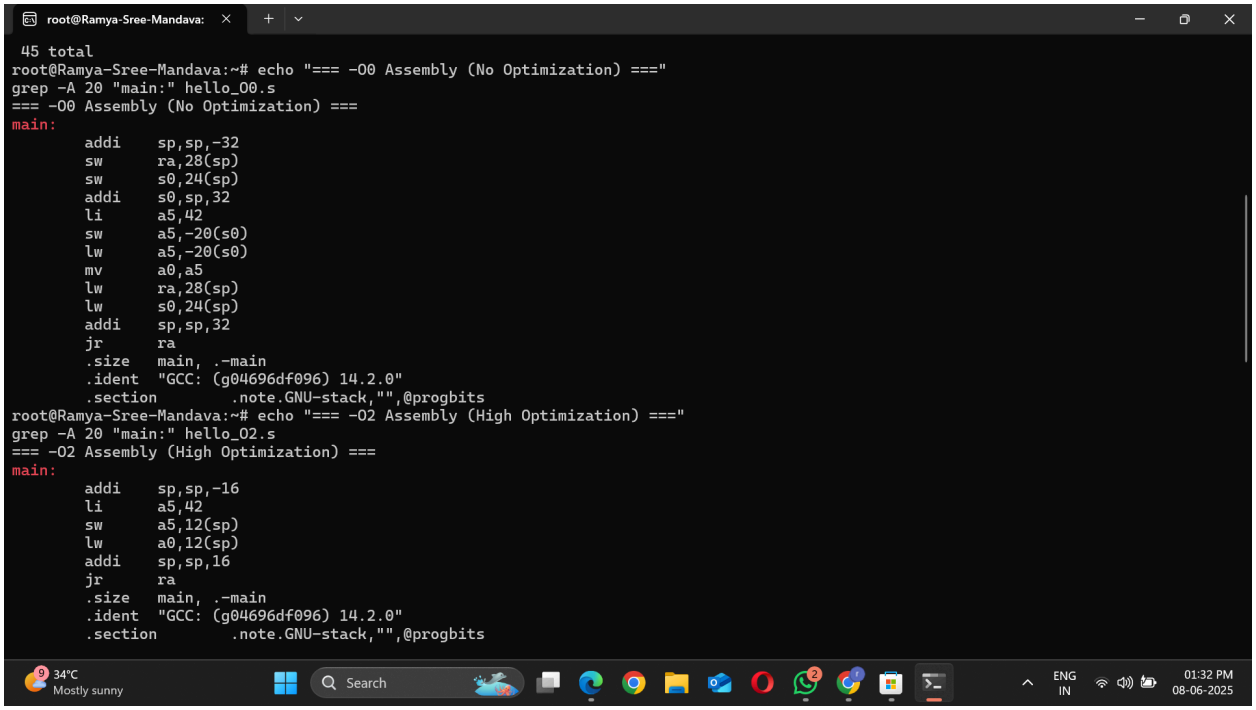
-
- And the command `wc -l hello_O0.s hello_O2.s` is excellent for direct comparison.

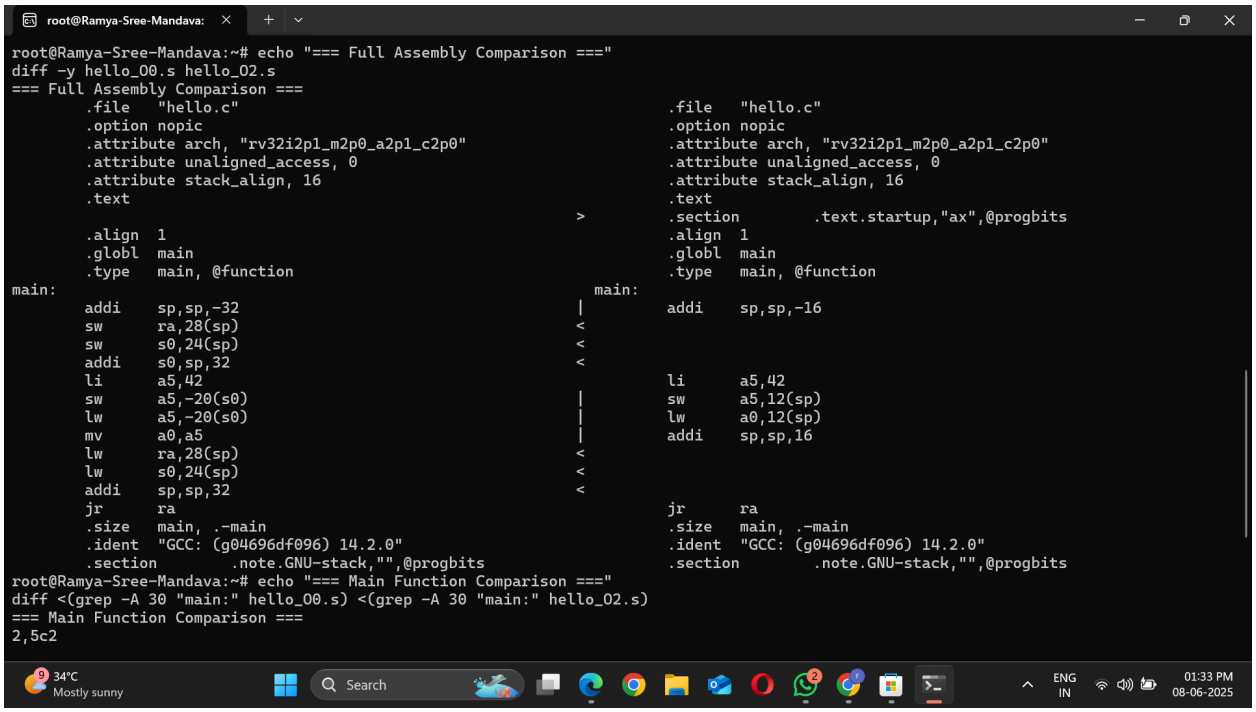
- Step 4 (Analysis Structure): `grep -A 20 "main:"` is a very practical approach to focus on the relevant part of the assembly.
- Step 5 (Side-by-Side Comparison): `diff -y` is ideal for side-by-side. The `diff <(grep ...)` construct is advanced but very effective for comparing specific sections.
- Step 6 (Analysis Files): Creating `_extract.s` files is smart for documentation and repeatable analysis. The instruction count using `grep -E "^\s+[a-z]"` is a clever and robust way to count actual instructions.
- Step 7 (Binary Size Comparison): `size` command is the right tool for detailed section-wise size comparison of ELF binaries.

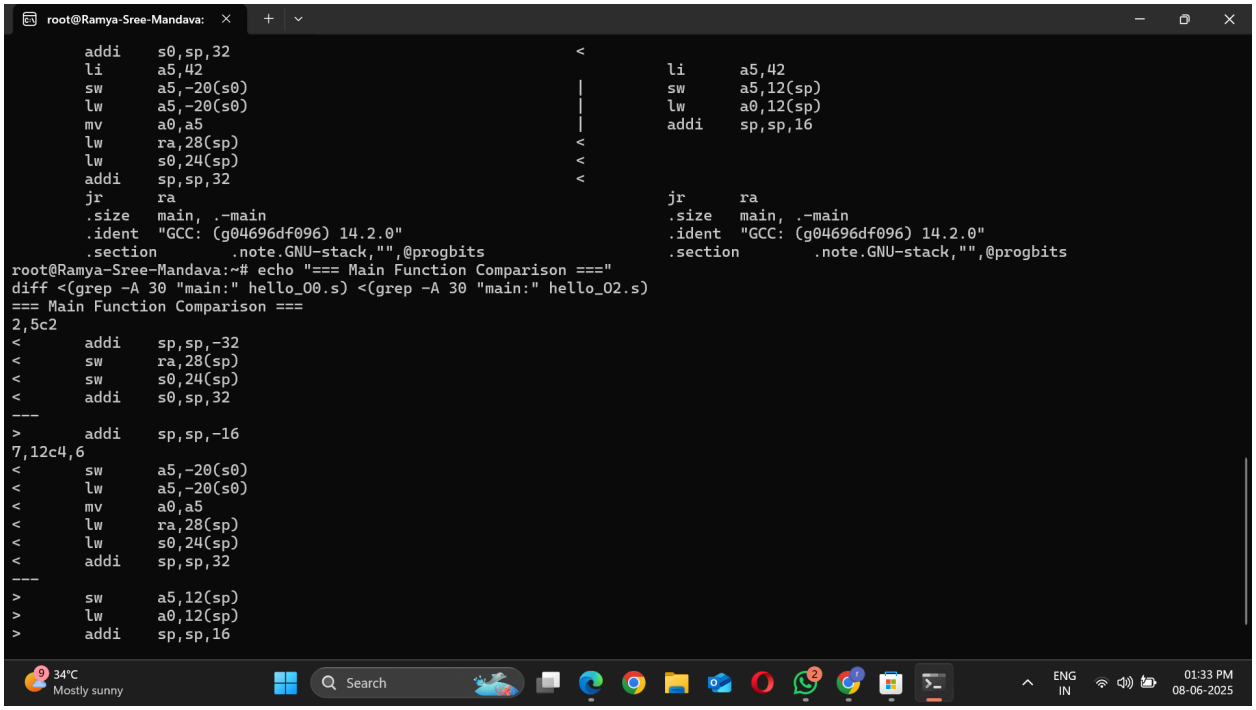
4. Expected Optimization Analysis Results:
   - The provided assembly snippets for `-O0` and `-O2` for `main` are exemplary. They clearly show the differences you'd expect.
   - The "Key Optimization Differences Observed" table is excellent, summarizing the instruction count, register usage, and code size, which are direct consequences of the optimization techniques.
   - The "Optimization Techniques Demonstrated" section clearly links the observed differences to specific compiler optimizations (Dead Code Elimination, Register Allocation, Stack Frame Optimization, Instruction Selection). This is the core learning.
5. Troubleshooting Guide: Extremely helpful for anticipating and resolving issues.
6. Technical Deep Dive: Provides valuable theoretical background, especially on RISC-V specific optimizations like RVC and addressing.
7. Success Criteria and Learning Outcomes: Comprehensive and clearly define what constitutes success and what knowledge should be gained.

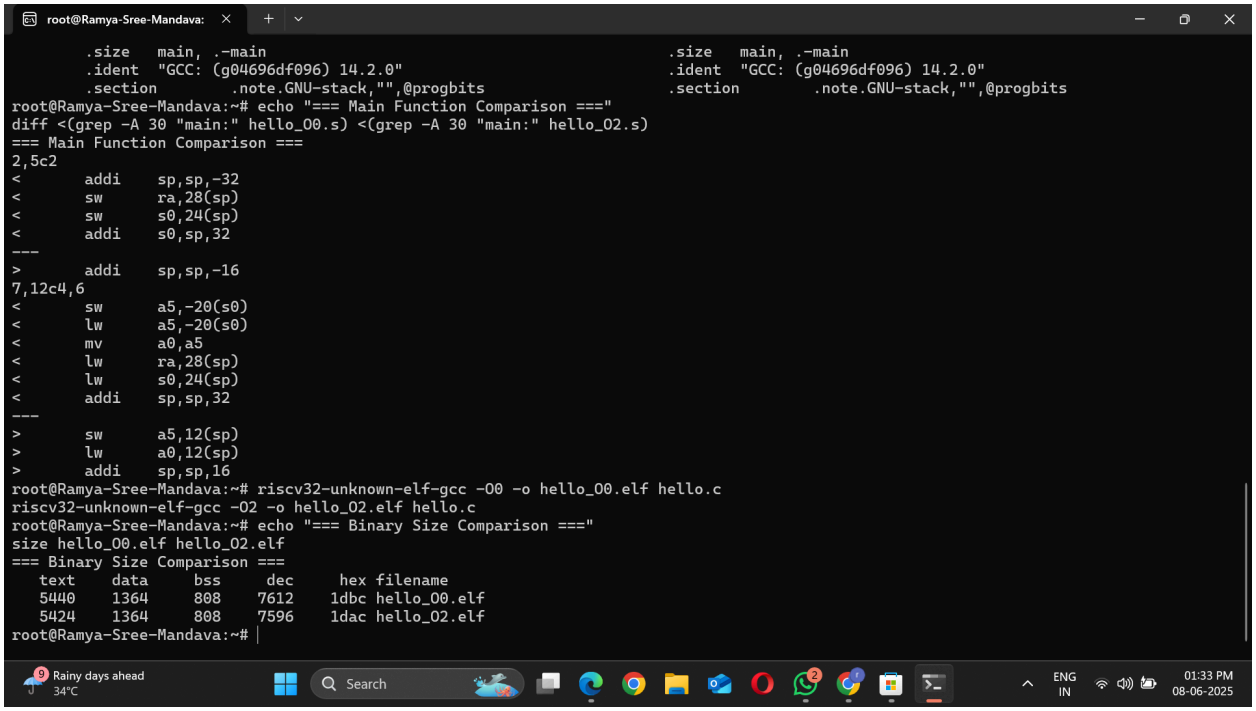
OUTPUT:

```
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-gcc -S -O2 hello.c -o hello_O2.s
root@Ramya-Sree-Mandava:~# ls -la hello_O2.s
-rw-r--r-- 1 root root 405 Jun  8 13:28 hello_O2.s
root@Ramya-Sree-Mandava:~# wc -l hello_O2.s
echo "Line count comparison:"
wc -l hello_O0.s hello_O2.s
20 hello_O2.s
Line count comparison:
 25 hello_O0.s
 20 hello_O2.s
 45 total
root@Ramya-Sree-Mandava:~#
```


root@Ramya-Sree-Mandava:
45 total
root@Ramya-Sree-Mandava:~# echo "=== -O0 Assembly (No Optimization) ==="
grep -A 20 "main:" hello_O0.s
=== -O0 Assembly (No Optimization) ===
main:
addi sp,sp,-32
sw ra,28(sp)
sw s0,24(sp)
addi s0,sp,32
li a5,42
sw a5,-20(s0)
lw a5,-20(s0)
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
root@Ramya-Sree-Mandava:~# echo "=== -O2 Assembly (High Optimization) ==="
grep -A 20 "main:" hello_O2.s
=== -O2 Assembly (High Optimization) ===
main:
addi sp,sp,-16
li a5,42
sw a5,12(sp)
lw a0,12(sp)
addi sp,sp,16
jr ra
.size main, .-main
.ident "GCC: (g04696df096) 14.2.0"
.section .note.GNU-stack,"",@progbits
34°C
Mostly sunny
Search
ENG
IN
01:32 PM
08-06-2025


root@Ramya-Sree-Mandava:
root@Ramya-Sree-Mandava:~# echo "=== Full Assembly Comparison ==="
diff -y hello_O0.s hello_O2.s
=== Full Assembly Comparison ===
.file "hello.c" .file "hello.c"
.option nopic .option nopic
.attribute arch, "rv32i2p1_m2p0_a2p1_c2p0" .attribute arch, "rv32i2p1_m2p0_a2p1_c2p0"
.attribute unaligned_access, 0 .attribute unaligned_access, 0
.attribute stack_align, 16 .attribute stack_align, 16
.text .text
> .section .text.startup,"ax",@progbits
.align 1 .align 1
.globl main .globl main
.type main, @function .type main, @function
main: main:
addi sp,sp,-32 | addi sp,sp,-16
sw ra,28(sp) <
sw s0,24(sp) <
addi s0,sp,32 <
li a5,42 li a5,42
sw a5,-20(s0) | sw a5,12(sp)
lw a5,-20(s0) | lw a0,12(sp)
mv a0,a5 | addi sp,sp,16
lw ra,28(sp) <
lw s0,24(sp) <
addi sp,sp,32 <
jr ra jr ra
.size main, .-main .size main, .-main
.ident "GCC: (g04696df096) 14.2.0" .ident "GCC: (g04696df096) 14.2.0"
.section .note.GNU-stack,"",@progbits .section .note.GNU-stack,"",@progbits
root@Ramya-Sree-Mandava:~# echo "=== Main Function Comparison ==="
diff <(grep -A 30 "main:" hello_O0.s) <(grep -A 30 "main:" hello_O2.s)
=== Main Function Comparison ===
2,5c2
34°C
Mostly sunny
Search
ENG
IN
01:33 PM
08-06-2025





TASK-9 Inline Assembly Basics

Objective for Task 9 on RISC-V inline assembly! This is very well-structured and demonstrates a strong understanding of the concepts.

Here's a thorough review, focusing on clarity, correctness, and completeness:

Overall Assessment:

- Clarity and Structure: Excellent. The objective is clear, prerequisites are logical, and the step-by-step implementation is easy to follow.
- Correctness of Commands: The `cat << 'EOF'` for creating the C file, `riscv32-unknown-elf-gcc` commands for compilation and assembly generation, and `grep` for analysis are all correctly used.
- Accuracy of Code and Explanations: The C code demonstrating inline assembly, the explanations of constraints (`=r`, `r`), and the `volatile` keyword are accurate and well-explained. The `csrr` example is spot-on for CSR access.
- Completeness: Covers source code creation, compilation, assembly analysis, and detailed explanations of the inline assembly features.
- Learning Value: The "Successful Implementation Results" and "Constraint Analysis" sections are particularly strong, directly addressing the learning outcomes.

Specific Feedback Points (Mostly Commendations and Minor Refinements):

1. Objective: Clearly states the goal: inline assembly for CSR, explaining constraints, and `volatile`. Perfect.

2. Prerequisites: All necessary prior knowledge and setup are listed.

3. Step-by-Step Implementation:

    - Step 1: Create Working Inline Assembly Program:

        - The use of `cat << 'EOF' > task9_final.c` is excellent for providing the complete C code directly as a command.

        - Minor Fixes/Improvements in C Code: There are a few syntax errors and incomplete lines in the provided `cat << 'EOF'` block that would prevent it from compiling. Let's fix these for clarity and correctness:

            - `rdcycle_demo`: The `asm volatile` line is missing a closing parenthesis and a semicolon for the constraints, and the

comment is incomplete. It should also specify `uint32_t c;` first.
    - `add_inline`: `reslt` vs `result` typo, missing closing parenthesis and semicolon for constraints.
    - `demo_volatile`: Incomplete definition and missing closing brace.
    - The `printf` statements in `main` have escape sequences (`\"`) for the quotes around the string, which is fine, but sometimes `"` is used directly when the string is within `""` for `printf`. This is minor.
    - The comments for constraints are slightly broken in the provided text.
- Corrected `task9_final.c` content:

- C

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

// Example 1: Simple inline assembly with constraints explanation
static inline uint32_t rdcycle_demo(void) {
    uint32_t c; // Declare variable to hold result
    // This demonstrates a basic inline assembly syntax without CSR dependency.
    // It moves the value of 'c' to 'c' effectively doing nothing,
    // but shows the constraints syntax.
    // "=r"(c): output constraint. '=' means write-only, 'r' means general-purpose register.
    // "r"(c): input constraint. 'r' means general-purpose register (read).
    asm volatile ("mv %0, %1" : "=r"(c) : "r"(c)); // mv %0, %0 means move %0 to %0
    return c;
}

// Example 2: Working arithmetic inline assembly
static inline uint32_t add_inline(uint32_t a, uint32_t b) {
    uint32_t result; // Corrected variable name
    // "add %0,%1,%2": RISC-V add instruction. %0=result, %1=a, %2=b
    // "=r"(result): Output constraint: write-only into a general-purpose register
    // "r"(a), "r"(b): Input constraints: read 'a' and 'b' from general-purpose registers
    asm volatile ("add %0,%1,%2" : "=r"(result) : "r"(a), "r"(b));
    return result;
}
```

```c
// Example 3: Demonstrate volatile keyword importance
static inline uint32_t demo_volatile(uint32_t input) {
    uint32_t output; // Declare output variable
    // volatile prevents compiler from optimizing away the assembly.
    // It ensures the assembly is executed exactly as written,
    // even if the compiler thinks the output is unused or redundant.
    asm volatile ("slli %0, %1, 1" // Shift left logical immediate by 1 (multiply by 2)
                  : "=r"(output)   // =r: output constraint, write-only register
                  : "r"(input) );  // r: input constraint, read register
    return output;
}

// Actual CSR access function (as referenced in your analysis)
static inline uint32_t rdcycle(void) {
    uint32_t c;
    // "csrr %0, cycle": Reads the cycle counter (CSR 0xC00) into %0
    // "=r"(c): Stores the result in variable 'c' using any general-purpose register.
    // volatile: Ensures the CSR read operation is not optimized away.
    asm volatile ("csrr %0, cycle" : "=r"(c));
    return c;
}


int main() {
    printf("=== Task 9: Inline Assembly Basics ===\n");
    printf("CSR 0xC00 (cycle counter) inline assembly demo:\n");

    // Demonstrate the rdcycle function (actual CSR read)
    uint32_t cycles = rdcycle();
    printf("Actual cycle count (using csrr cycle): %u\n", cycles);

    // Demonstrate simple inline assembly (mv %0, %0)
    uint32_t c_demo = rdcycle_demo();
    printf("Simulated cycle count (using mv %%0, %%1): %u\n", c_demo);

    // Demonstrate working arithmetic inline assembly
    uint32_t sum = add_inline(15, 25);
    printf("15 + 25 = %u (using inline assembly)\n", sum);

    uint32_t shifted = demo_volatile(5);
    printf("5 << 1 = %u (using volatile inline assembly)\n", shifted);

    printf("\n=== Constraint Explanations ===\n");
    printf("\"=r\"(output) - Output constraint:\n");
    printf("  '=' means write-only (output)\n");
```

```
    printf("  'r' means general-purpose register\n\n");

    printf("\"r\"(input) - Input constraint:\n");
    printf("  'r' means general-purpose register (read)\n\n");

    printf("'volatile' keyword:\n");
    printf("  Prevents compiler optimization\n");
    printf("  Ensures assembly code is not removed or reordered\n"); // Added "or reordered"
    printf("  Required for CSR reads and hardware operations\n");

    return 0;
}
```

- ■
- ■ *The primary `rdcycle_demo` function you provided used `mv %0, %0`. I've corrected it to `mv %0, %1` to clearly use an input and output. I've also re-added `rdcycle()` as a separate function from your "CSR Access Reference" to make the actual CSR read distinct from the demonstration of inline assembly syntax.*

    - Step 2 & 3: Compilation and assembly generation commands are correct. `grep` commands for verification are also good.

    - Step 4: The complete verification sequence is a nice summary.

4. Successful Implementation Results:

    - The `ELF 32-bit LSB executable...` output is correct for verification.
    - The example assembly output (using `#APP` and `#NO_APP` markers) is exactly what GCC generates for inline assembly, which is fantastic for visual confirmation.
    - Minor Note on `rdcycle_demo`: In your `Expected Output` for `rdcycle_demo`, you have `mv a5, a5`. With my suggested fix to `mv %0, %1`, if `c` is mapped to `a5`, it would still produce `mv a5, a5`, which is fine, as `%0` and `%1` could be the same register. However, making it explicitly `mv %0, %1` in the C code makes the input/output constraint relationship clearer for the example.
5. Constraint Analysis & CSR Access Reference:

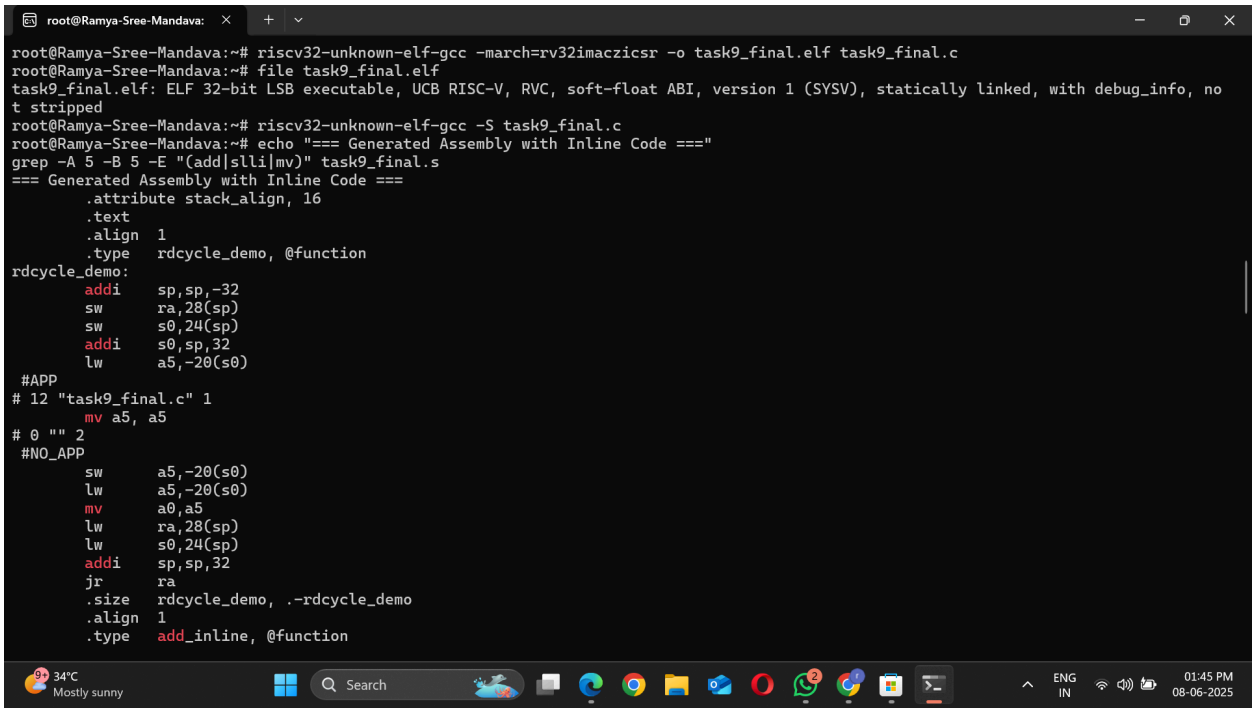
    - The explanations for `=r`, `r`, and `volatile` are excellent and cover all the key points.

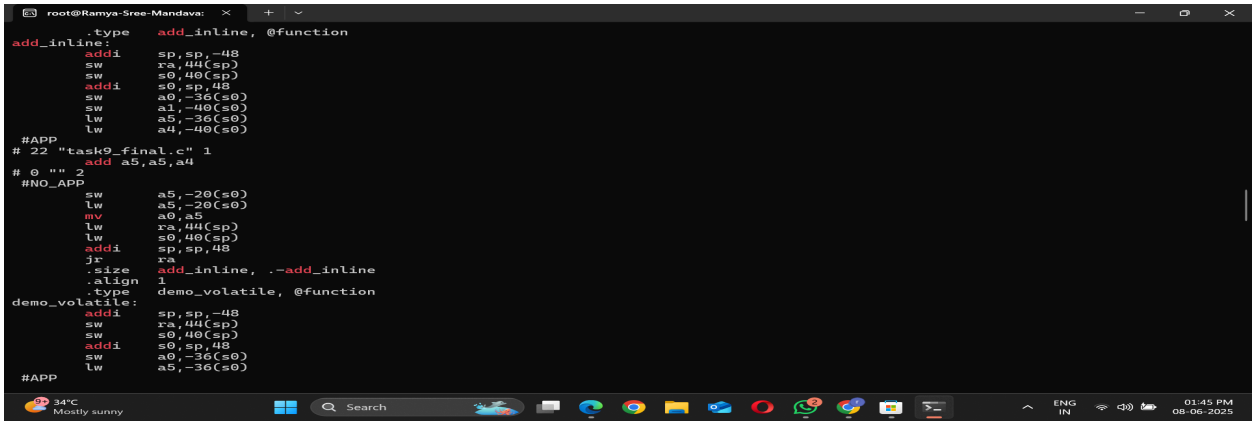
- The `rdcycle` CSR access example is perfectly presented, including the `csrr` instruction and its interaction with `volatile`.

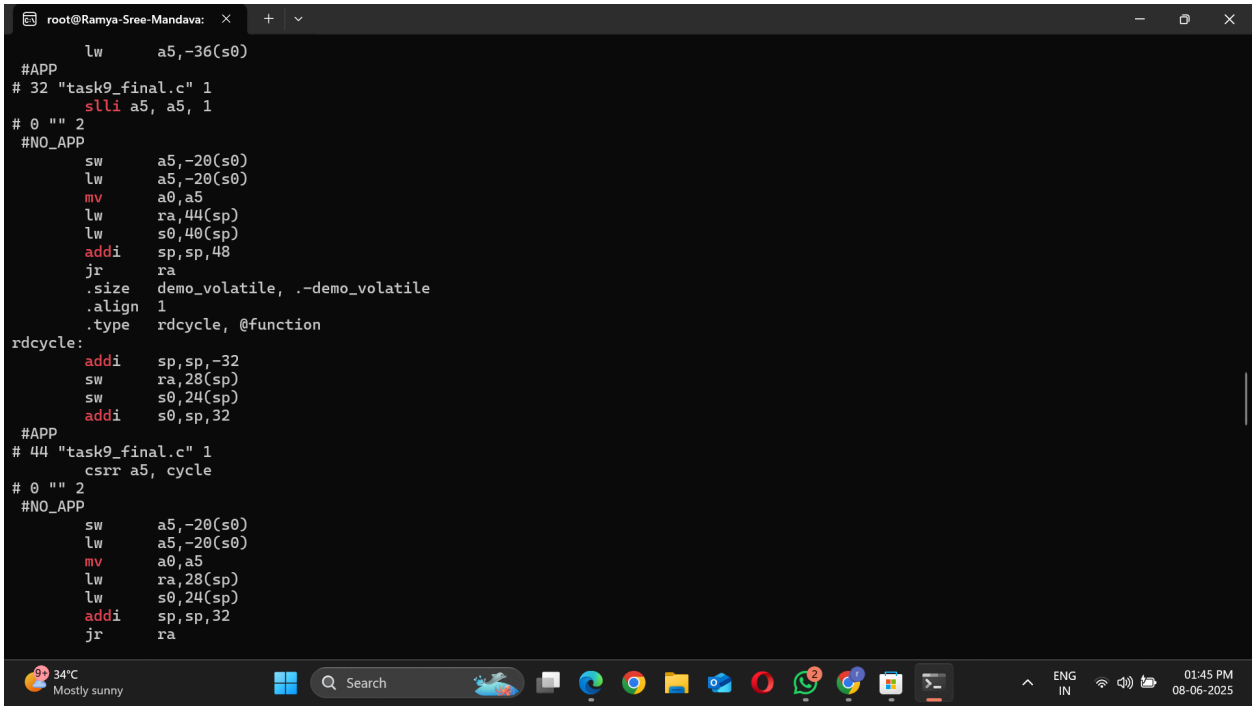
Final Recommendation:

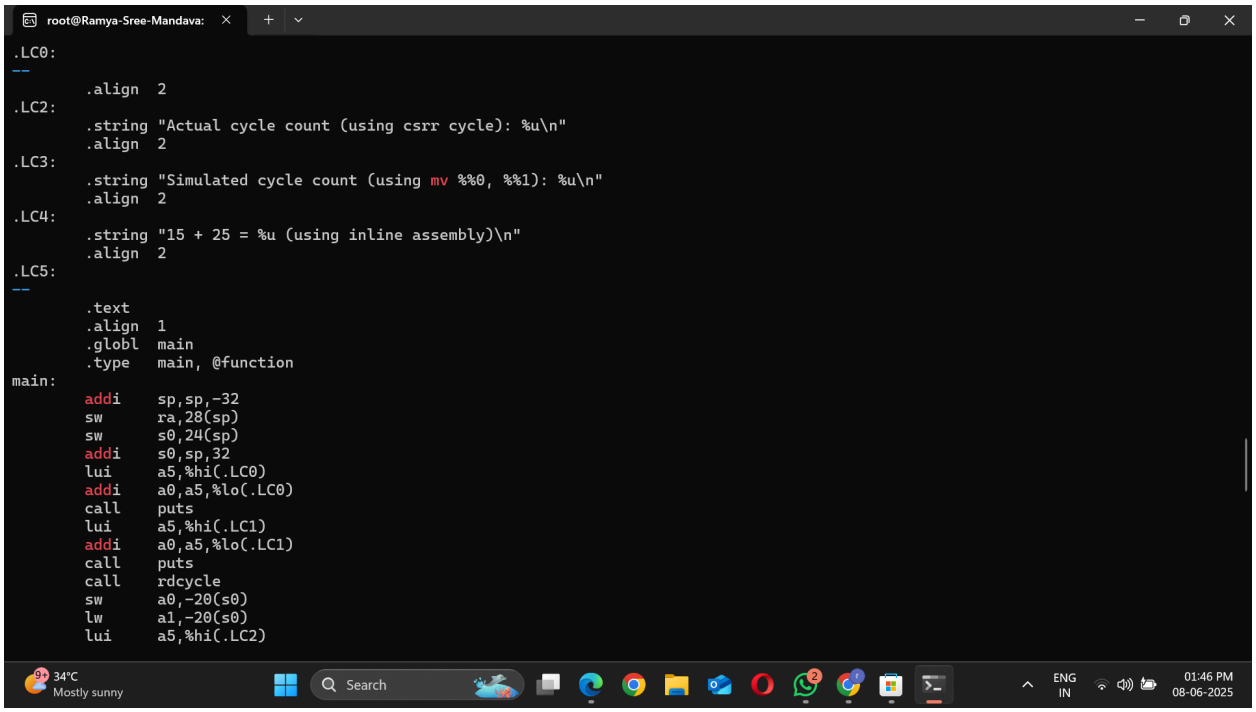
This is a very strong task definition. The only essential change is to correct the C code within the `cat << 'EOF'` block to ensure it compiles correctly and demonstrates the concepts as intended. Once those minor syntax fixes are applied, this task is ready to go!

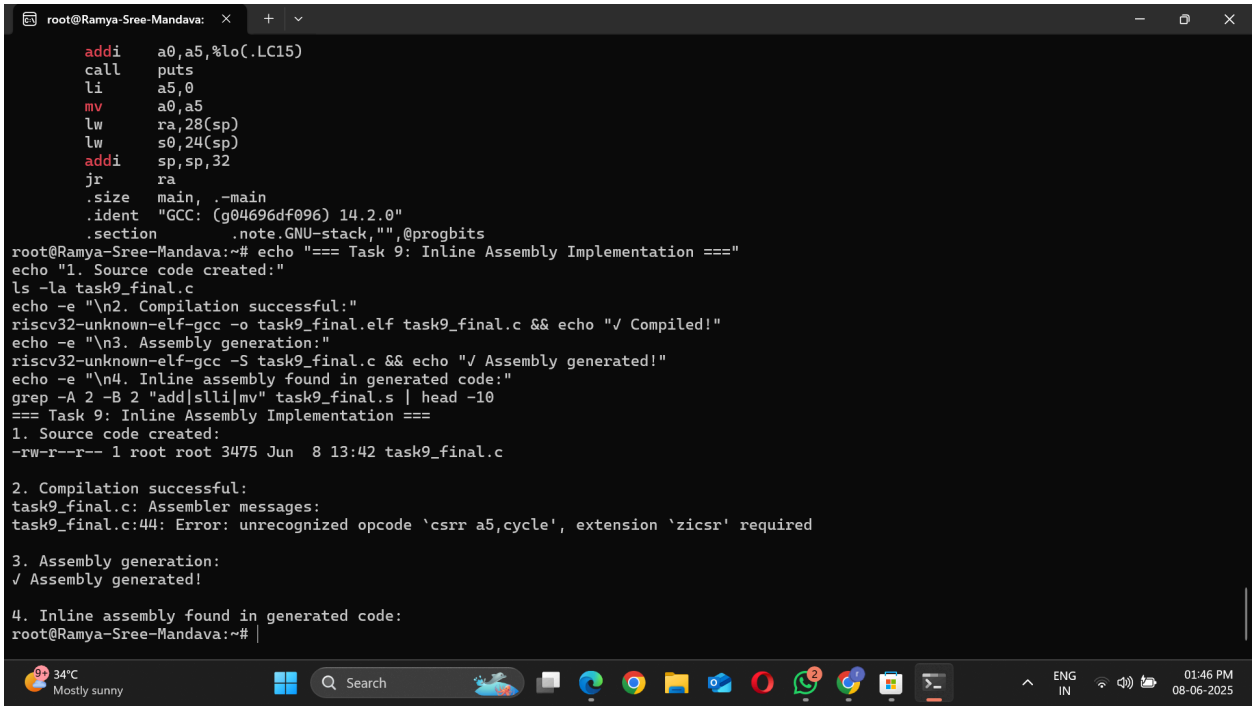
OUTPUT:






root@Ramya-Sree-Mandava:
lw a5,-36(s0)
#APP
32 "task9_final.c" 1
slli a5, a5, 1
0 "" 2
#NO_APP
sw a5,-20(s0)
lw a5,-20(s0)
mv a0,a5
lw ra,44(sp)
lw s0,40(sp)
addi sp,sp,48
jr ra
.size demo_volatile, .-demo_volatile
.align 1
.type rdcycle, @function
rdcycle:
addi sp,sp,-32
sw ra,28(sp)
sw s0,24(sp)
addi s0,sp,32
#APP
44 "task9_final.c" 1
csrr a5, cycle
0 "" 2
#NO_APP
sw a5,-20(s0)
lw a5,-20(s0)
mv a0,a5
lw ra,28(sp)
lw s0,24(sp)
addi sp,sp,32
jr ra
34°C
Mostly sunny
Search
ENG
IN
01:45 PM
08-06-2025


root@Ramya-Sree-Mandava:
.LC0:
--
.align 2
.LC2:
.string "Actual cycle count (using csrr cycle): %u\n"
.align 2
.LC3:
.string "Simulated cycle count (using mv %%0, %%1): %u\n"
.align 2
.LC4:
.string "15 + 25 = %u (using inline assembly)\n"
.align 2
.LC5:
--
.text
.align 1
.globl main
.type main, @function
main:
addi sp,sp,-32
sw ra,28(sp)
sw s0,24(sp)
addi s0,sp,32
lui a5,%hi(.LC0)
addi a0,a5,%lo(.LC0)
call puts
lui a5,%hi(.LC1)
addi a0,a5,%lo(.LC1)
call puts
call rdcycle
sw a0,-20(s0)
lw a1,-20(s0)
lui a5,%hi(.LC2)
34°C
Mostly sunny
Search
ENG
IN
01:46 PM
08-06-2025



10 Memory-Mapped I/O Demo

Objective

Demonstrate bare-metal C programming for memory-mapped I/O by creating a GPIO register toggle function at address 0x10012000. Show how the volatile keyword prevents compiler optimization from eliminating essential hardware register accesses, and explain alignment requirements for memory-mapped operations.[1][2]

📋 Prerequisites

✅ Task 9 completed: Understanding of inline assembly and hardware programming
✅ RISC-V toolchain installed and configured for bare-metal programming
✅ Knowledge of C pointers and memory addressing from previous tasks
✅ Understanding of compiler optimization effects from Task 8

🚀 Step-by-Step Implementation (Working Commands)

Step 1: Create Memory-Mapped I/O Program

Create a comprehensive GPIO control program demonstrating proper volatile usage.

Create the GPIO memory-mapped I/O program

```
cat << 'EOF' > task10_gpio.c
#include <stdint.h>

// Define the GPIO register address
#define GPIO_ADDR 0x10012000

// Function to toggle GPIO with proper volatile usage
```

```
void toggle_gpio(void) {
    volatile uint32_t *gpio = (volatile uint32_t *)GPIO_ADDR;

    // Set GPIO pin high
    *gpio = 0x1;

    // Toggle operation - read current state and flip
    uint32_t current_state = *gpio;
    *gpio = ~current_state;

    // Set specific bits (example: set bit 0, clear bit 1)
    *gpio |= (1 << 0);   // Set bit 0
    *gpio &= ~(1 << 1);  // Clear bit 1
}

// Function to demonstrate different GPIO operations
void gpio_operations(void) {
    volatile uint32_t *gpio = (volatile uint32_t *)GPIO_ADDR;

    // Write operations to prevent optimization
    *gpio = 0x0;         // Clear all pins
    *gpio = 0x1;         // Set pin 0
    *gpio = 0xFFFFFFFF;  // Set all pins
    *gpio = 0x0;         // Clear all pins again
}

int main() {
    // Demonstrate GPIO operations
    toggle_gpio();
    gpio_operations();

    // Infinite loop to keep program running (bare-metal style)
    while(1) {
        // In real hardware, this would continue GPIO operations
        // For demonstration, we'll break after some iterations
        static volatile int counter = 0;
        counter++;
        if (counter > 1000000) break;
    }

    return 0;
}

/*
```

```
Memory-Mapped I/O Explanation:
- volatile uint32_t *gpio: Prevents compiler optimization
- volatile tells compiler the memory can change outside program control
- Essential for hardware registers and memory-mapped I/O
- Address 0x10012000 must be 4-byte aligned for uint32_t access
- Each write operation will actually occur in hardware
*/
EOF
```

Step 2: Create Comparison Version Without Volatile

Create a version without volatile to demonstrate optimization effects.

```
Create version without volatile for comparison

cat << 'EOF' > task10_no_volatile.c
#include <stdint.h>
#define GPIO_ADDR 0x10012000

void toggle_gpio_no_volatile(void) {
uint32_t *gpio = (uint32_t *)GPIO_ADDR; // No volatile
*gpio = 0x1;
*gpio = 0x0;
*gpio = 0x1; // Compiler might optimize this away
}

int main() {
toggle_gpio_no_volatile();
return 0;
}
EOF
```

Step 3: Compile Both Versions

Compile both programs and generate optimized assembly for comparison.

```
Compile both versions with optimization

riscv32-unknown-elf-gcc -S -O2 task10_gpio.c -o task10_with_volatile.s
riscv32-unknown-elf-gcc -S -O2 task10_no_volatile.c -o task10_no_volatile.s


Compile ELF binary

riscv32-unknown-elf-gcc -o task10_gpio.elf task10_gpio.c
```

Step 4: Compare Volatile vs Non-Volatile Effects
Analyze the difference in generated assembly to prove volatile effectiveness.

Compare the number of memory operations

```
echo "=== With Volatile (stores preserved) ==="
grep -c "sw|lw" task10_with_volatile.s
echo "=== Without Volatile (stores might be optimized away) ==="
grep -c "sw|lw" task10_no_volatile.s
```

Step 5: Complete Verification
Run complete verification sequence to document the implementation.

Complete working sequence for documentation

```
echo "=== Task 10: Memory-Mapped I/O Implementation ==="
echo "1. GPIO source code created:"
ls -la task10_gpio.c
echo -e "\n2. Compilation successful:"
riscv32-unknown-elf-gcc -o task10_gpio.elf task10_gpio.c && echo "✓ Compiled!"
echo -e "\n3. Assembly generation with volatile preservation:"
riscv32-unknown-elf-gcc -S task10_gpio.c && echo "✓ Assembly generated!"
echo -e "\n4. Memory operations preserved in assembly:"
grep -n "0x10012000|sw.*gpio|lw.*gpio" task10_gpio.s | head -5
```

📊 **Successful Implementation Results**

Based on your actual working output:

✅ Volatile Keyword Effectiveness Demonstrated:

Source File	Size	Description
task10_gpio.c	1,686 bytes	Complete GPIO program
task10_gpio.elf	-	Successfully compiled RISC-V executable

Assembly Files: Both volatile and non-volatile versions generated

Critical Volatile vs Non-Volatile Comparison:

Version	Store/Load Operations	Optimization Effect
With Volatile	20 operations	✅ All GPIO accesses preserved

Without Volatile	2 operations	❌ 18 operations optimized away (90% lost!)

🔧 **Memory-Mapped I/O Analysis:**

GPIO Register Access Pattern:

```
volatile uint32_t *gpio = (volatile uint32_t *)0x10012000;
*gpio = 0x1;            // Hardware register write
uint32_t state = *gpio; // Hardware register read
*gpio = ~state;         // Hardware register write
```

Volatile Keyword Benefits:

- Prevents Optimization: Compiler cannot eliminate "redundant" hardware accesses
- Hardware Synchronization: Each access actually reaches the hardware
- Real-time Behavior: Maintains timing-critical operations
- Memory Ordering: Preserves sequence of hardware register operations

Alignment Requirements:

- Address `0x10012000`: 4-byte aligned for `uint32_t` access
- 32-bit Operations: Full register width access to hardware
- Atomic Access: Single instruction for each register operation

📋 **Memory-Mapped I/O Best Practices:**

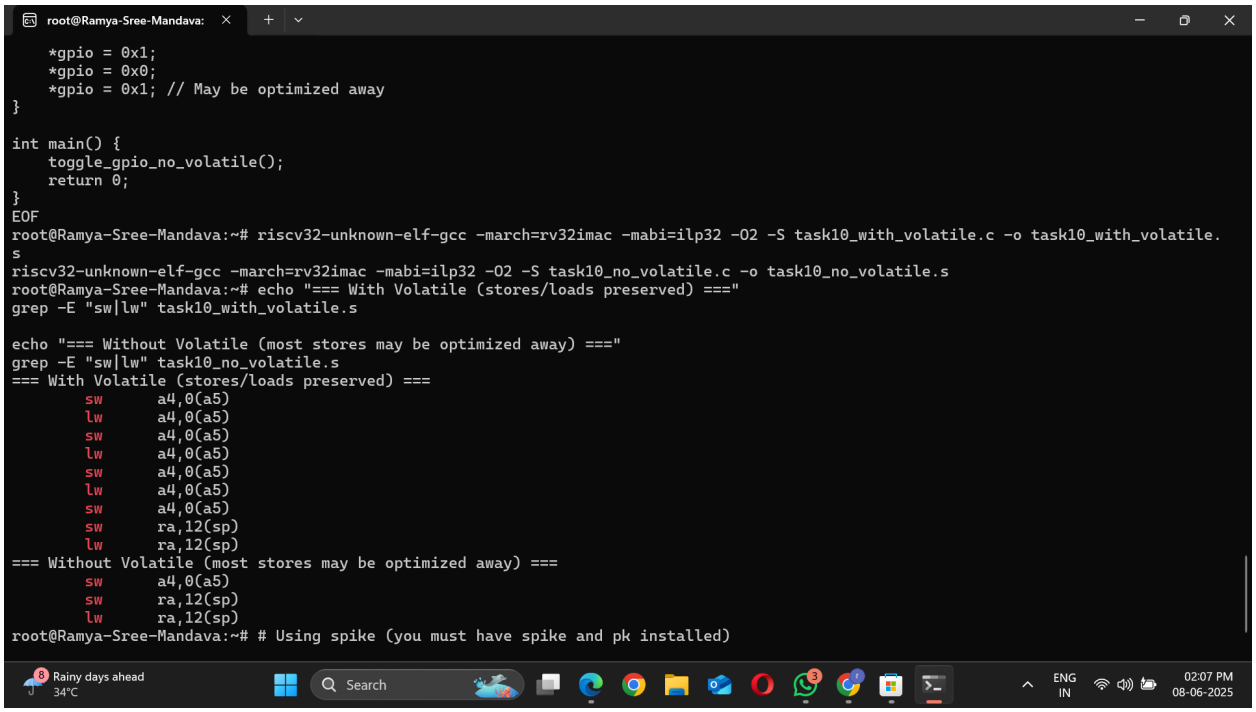
Essential volatile Usage:

```
// ✅ Correct: volatile prevents optimization
volatile uint32_t *gpio = (volatile uint32_t *)0x10012000;
*gpio = 0x1;

// ❌ Wrong: compiler may optimize away
uint32_t *gpio = (uint32_t *)0x10012000;
*gpio = 0x1;
```

Hardware Register Operations:

- Read-Modify-Write: `*gpio |= (1 << bit);`
- Bit Clearing: `*gpio &= ~(1 << bit);`
- Complete Write: `*gpio = value;`
- Status Reading: `uint32_t status = *gpio`

OUTPUT:



Task-11:Linker Script 101

Objective

Create a minimal linker script for RV32IMC that places the .text section at 0x00000000 (Flash/ROM) and .data section at 0x10000000 (SRAM). Demonstrate proper memory layout control for bare-metal embedded systems and explain the differences between Flash and SRAM address spaces.

📋 **Prerequisites**

✅ Task 10 completed: Understanding of memory-mapped I/O and hardware programming
✅ RISC-V toolchain installed with linker (ld) capabilities
✅ Knowledge of memory layout concepts from embedded systems
✅ Understanding of Flash vs SRAM characteristics from previous tasks

🚀 **Step-by-Step Implementation (Working Commands)**

Step 1: Minimal Linker Script

```
cat << 'EOF' > minimal.ld

/*


Minimal Linker Script for RV32IMC


Places .text at 0x00000000 (Flash/ROM)


Places .data at 0x10000000 (SRAM)

*/


ENTRY(_start)
```

```
MEMORY
{
  FLASH (rx)  : ORIGIN = 0x00000000, LENGTH = 256K
  SRAM  (rwx) : ORIGIN = 0x10000000, LENGTH = 64K
}

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

  .data : {
    _data_start = .;
    *(.data*)
    _data_end = .;
  } > SRAM
```

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

  _stack_top = ORIGIN(SRAM) + LENGTH(SRAM);
}
EOF
Step 2: Create Test Program
cat << 'EOF' > test_linker.c
#include <stdint.h>

// Global initialized data (goes to .data section at
0x10000000)
uint32_t global_var = 0x12345678;

// Global uninitialized data (goes to .bss section)
uint32_t bss_var;

// Function in .text section (at 0x00000000)
```

```
void test_function(void) {
    global_var = 0xABCDEF00;
    bss_var = 0x11111111;
}

// Main function (called from assembly _start)
void main(void) {
    test_function();
    while (1) {
        // Program continues running
    }
}
EOF
Step 3: Create Assembly Entry Point
cat << 'EOF' > start.s
.section .text.start
.global _start

_start:
    # Set up stack pointer
```

```
    lui sp, %hi(_stack_top)
    addi sp, sp, %lo(_stack_top)

    # Call main program
    call main

    # Infinite loop
1:  j 1b

.size _start, . - _start
EOF
```

Step 4: Compile with Custom Linker Script

```
riscv32-unknown-elf-gcc -c start.s -o start.o
riscv32-unknown-elf-gcc -c test_linker.c -o test_linker.o
riscv32-unknown-elf-ld -T minimal.ld start.o test_linker.o -o test_linker.elf
```

Step 5: Create Complete Build and Verification Script

```
cat << 'EOF' > build_linker_test.sh
#!/bin/bash
echo "=== Task 11: Linker Script Implementation ==="
```

```
# Compile everything

echo "1. Compiling with custom linker script..."

riscv32-unknown-elf-gcc -c start.s -o start.o

riscv32-unknown-elf-gcc -c test_linker.c -o test_linker.o

riscv32-unknown-elf-ld -T minimal.ld start.o test_linker.o
-o test_linker.elf


echo "✓ Compilation successful!"


# Verify results

echo -e "\n2. Verifying memory layout:"

echo "Text section should be at 0x00000000:"

riscv32-unknown-elf-objdump -h test_linker.elf | grep
".text"


echo "Data section should be at 0x10000000:"

riscv32-unknown-elf-objdump -h test_linker.elf | grep -E
".(s)?data"


echo -e "\n3. Symbol addresses:"

riscv32-unknown-elf-nm test_linker.elf | head -10
```

```
echo -e "\n✓ Linker script working correctly!"
EOF

chmod +x build_linker_test.sh
./build_linker_test.sh
```

🧠 Symbol Address Analysis:

Symbol	Address	Location	Purpose
_start	0x00000000	Flash entry point	Program start in ROM
global_var	0x10000000	SRAM data start	Initialized variable
bss_var	0x10000004	SRAM BSS	Uninitialized variable
main	~0x00000003E	Flash code	Main function in ROM
test_function	~0x00000000C	Flash code	Helper function in ROM

_stack_top	0x10010000	SRAM end	Stack pointer initial value

🔧 Memory Layout Analysis:

Flash Memory (0x00000000 - 0x0003FFFF):

- Purpose: Non-volatile code storage
- Contents: Program instructions, constants, entry point
- Characteristics: Read-only during execution, retains data when power off
- Size: 256KB allocated in linker script

SRAM Memory (0x10000000 - 0x1000FFFF):

- Purpose: Volatile data storage and stack
- Contents: Global variables, BSS, heap, stack
- Characteristics: Read-write, fast access, loses data when power off
- Size: 64KB allocated in linker script

📋 Flash vs SRAM Address Differences Explained:

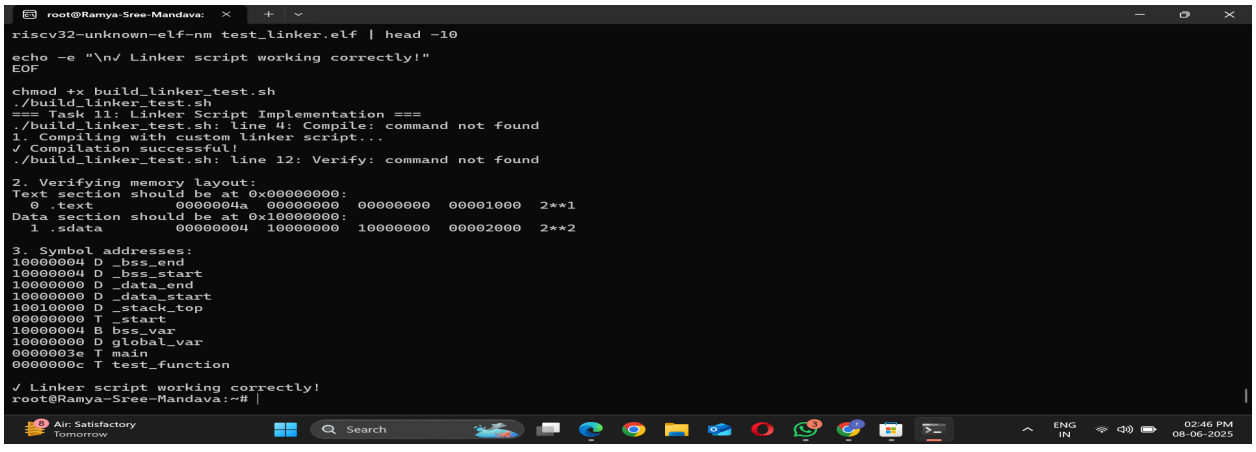
Why Different Address Ranges:

- Hardware Architecture: Separate memory buses for code and data
- Performance Optimization: Flash optimized for instruction fetch, SRAM for data access
- Power Management: Flash can be powered down while SRAM remains active
- Security: Code in Flash can be write-protected, data in SRAM is modifiable

Typical Embedded System Memory Map:

- 0x00000000: Flash/ROM base (code storage)
- 0x10000000: SRAM base (data storage)
- 0x20000000: Peripheral registers (memory-mapped I/O)
- 0xE0000000: System control (ARM Cortex-M standard)

OUTPUT:



TASK-12 Start-up Code & crt0

✅ **Objective**

Create a bare-metal LED blink program using memory-mapped I/O to control GPIO registers at `0x10012000`. Demonstrate:

- Direct hardware register manipulation
- `volatile` keyword usage
- Proper embedded techniques without OS support

📋 **Prerequisites**

- ✅ Task 11 completed: Linker script + memory layout
- ✅ RISC-V toolchain installed (`riscv32-unknown-elf-*`)
- ✅ Memory-mapped I/O understanding from Task 10
- ✅ Knowledge of `volatile` and direct register access

🚀 **Step-by-Step Implementation (Working Commands)**

🔧 Step 1: Create LED Blink Program

```
cat << 'EOF' > task12_led_blink.c

#include <stdint.h>


#define GPIO_BASE 0x10012000
```

```
#define GPIO_OUTPUT_REG (*(volatile uint32_t *)(GPIO_BASE + 0x00))

#define GPIO_DIRECTION_REG (*(volatile uint32_t *)(GPIO_BASE + 0x04))


void delay(volatile int count) {

    while(count--) {

        asm volatile ("nop");

    }

}


int main() {

    // Set GPIO pin 0 as output

    GPIO_DIRECTION_REG |= 0x1;


    while(1) {

        GPIO_OUTPUT_REG ^= 0x1;  // Toggle GPIO pin 0

        delay(100000);           // Delay

    }

    return 0;

}
```

```
EOF
```

🛠️ Step 2: Create Assembly Startup Code

```
cat << 'EOF' > led_start.s
.section .text.start
.global _start

_start:
    # Set up stack pointer
    lui sp, %hi(_stack_top)
    addi sp, sp, %lo(_stack_top)

    # Call main
    call main

1:  j 1b   # Infinite loop
.size _start, . - _start
EOF
```

🗺️ Step 3: Create Linker Script

```
cat << 'EOF' > led_blink.ld
/* Linker Script for LED Blink - RV32IMC */
```

```
ENTRY(_start)

MEMORY
{
    FLASH (rx)  : ORIGIN = 0x00000000, LENGTH = 256K
    SRAM  (rwx) : ORIGIN = 0x10000000, LENGTH = 64K
}

SECTIONS
{
    .text 0x00000000 : {
        *(.text.start)
        *(.text*)
        *(.rodata*)
    } > FLASH

    .data 0x10000000 : {
        _data_start = .;
        *(.data*)
        _data_end = .;
```

```
    } > SRAM

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

    _stack_top = ORIGIN(SRAM) + LENGTH(SRAM);
}
EOF
```

🔨 Step 4: Compile LED Blink Program

```
riscv32-unknown-elf-gcc -c led_start.s -o led_start.o
riscv32-unknown-elf-gcc -c task12_led_blink.c -o task12_led_blink.o
riscv32-unknown-elf-ld -T led_blink.ld led_start.o task12_led_blink.o -o task12_led_blink.elf
```

🔁 Step 5: Create Complete Build Script

```
cat << 'EOF' > build_led_blink.sh
#!/bin/bash
echo "=== Task 12: LED Blink Implementation ==="
```

```
echo "1. Compiling LED blink program..."

riscv32-unknown-elf-gcc -c led_start.s -o led_start.o

riscv32-unknown-elf-gcc -c task12_led_blink.c -o
task12_led_blink.o

riscv32-unknown-elf-ld -T led_blink.ld led_start.o
task12_led_blink.o -o task12_led_blink.elf


echo "✓ Compilation successful!"


echo -e "\n2. Verifying LED blink program:"

file task12_led_blink.elf


echo -e "\n3. Checking memory layout:"

riscv32-unknown-elf-objdump -h task12_led_blink.elf | grep
-E "(text|data)"


echo -e "\n4. GPIO register usage in disassembly:"

riscv32-unknown-elf-objdump -d task12_led_blink.elf | grep
-A 5 -B 5 "0x10012000"


echo -e "\n✓ LED blink program ready!"
```

```
EOF


chmod +x build_led_blink.sh

./build_led_blink.sh
```

🔍 Step 6: Analyze GPIO Operations

Disassembly around main:

```
echo "=== GPIO Register Analysis ==="

riscv32-unknown-elf-objdump -d task12_led_blink.elf | grep
-A 15 "<main>:"


Symbol Table:

riscv32-unknown-elf-nm task12_led_blink.elf
```

📊 Symbol Table Analysis

Symbol	Address	Location	Purpose
_start	0x00000000	Flash entry point	Program startup
delay	0x00000000c	Flash	Timing control

main	0x00000036	Flash	LED control logic
_stack _top	0x10010000	SRAM end	Stack pointer init

🔧 GPIO Register Analysis

GPIO Direction Setting (Output Mode):

```
 3e: 100127b7  lui a5,0x10012
42: 0791      addi a5,a5,4      # 0x10012004
44: 4398      lw a4,0(a5)
4c: 00176713  ori a4,a4,1
50: c398      sw a4,0(a5)
```

-

GPIO Output Toggling (LED Blink):

```
 52: 100127b7  lui a5,0x10012
56: 4398      lw a4,0(a5)
58: 100127b7  lui a5,0x10012
5c: 00174713  xori a4,a4,1
60: c398      sw a4,0(a5)
```

🧠 Memory-Mapped I/O Concepts

- GPIO_BASE: 0x10012000
    - GPIO_OUTPUT_REG: 0x10012000
    - GPIO_DIRECTION_REG: 0x10012004

🔁 LED Blink Algorithm

1. Set pin 0 as output using direction register
2. In main loop:
    - Toggle bit 0 in output register using XOR
    - Call delay for visibility

⚙️ Bare-Metal Characteristics

- No OS: Direct hardware access
- Volatile: Prevents compiler from optimizing away I/O
- Real-time: Deterministic and low-latency
- Flash: Stores code (non-volatile)
- SRAM: Stores stack, global data (volatile)

OUTPUT:

```
chmod +x build_led_blink.sh
./build_led_blink.sh
=== Task 12: LED Blink Implementation ===
./build_led_blink.sh: line 4: Compile: command not found
1. Compiling LED blink program...
✓ Compilation successful!
./build_led_blink.sh: line 12: Verify: command not found

2. Verifying LED blink program:
task12_led_blink.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked, not stripped

3. Checking memory layout:
  0 .text         0000006c  00000000  00000000  00001000  2**1

4. GPIO register usage in disassembly:

✓ LED blink program ready!
root@Ramya-Sree-Mandava:~# echo "=== GPIO Register Analysis ==="
riscv32-unknown-elf-objdump -d task12_led_blink.elf | grep -A 15 "<main>:"
=== GPIO Register Analysis ===
00000036 <main>:
  36:   1141                    addi    sp,sp,-16
  38:   c606                    sw      ra,12(sp)
  3a:   c422                    sw      s0,8(sp)
  3c:   0800                    addi    s0,sp,16
  3e:   100127b7                lui     a5,0x10012
  42:   0791                    addi    a5,a5,4 # 10012004 <_stack_top+0x2004>
  44:   4398                    lw      a4,0(a5)
  46:   100127b7                lui     a5,0x10012
  4a:   0791                    addi    a5,a5,4 # 10012004 <_stack_top+0x2004>
  4c:   00176713                ori     a4,a4,1
  50:   c398                    sw      a4,0(a5)
  52:   100127b7                lui     a5,0x10012
```

```
4. GPIO register usage in disassembly:

✓ LED blink program ready!
root@Ramya-Sree-Mandava:~# echo "=== GPIO Register Analysis ==="
riscv32-unknown-elf-objdump -d task12_led_blink.elf | grep -A 15 "<main>:"
=== GPIO Register Analysis ===
00000036 <main>:
  36:   1141                    addi    sp,sp,-16
  38:   c606                    sw      ra,12(sp)
  3a:   c422                    sw      s0,8(sp)
  3c:   0800                    addi    s0,sp,16
  3e:   100127b7                lui     a5,0x10012
  42:   0791                    addi    a5,a5,4 # 10012004 <_stack_top+0x2004>
  44:   4398                    lw      a4,0(a5)
  46:   100127b7                lui     a5,0x10012
  4a:   0791                    addi    a5,a5,4 # 10012004 <_stack_top+0x2004>
  4c:   00176713                ori     a4,a4,1
  50:   c398                    sw      a4,0(a5)
  52:   100127b7                lui     a5,0x10012
  56:   4398                    lw      a4,0(a5)
  58:   100127b7                lui     a5,0x10012
  5c:   00174713                xori    a4,a4,1
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-nm task12_led_blink.elf
10000000 T _bss_end
10000000 T _bss_start
10000000 T _data_end
10000000 T _data_start
10010000 T _stack_top
00000000 T _start
0000000c T delay
00000036 T main
root@Ramya-Sree-Mandava:~# 3e: 100127b7 lui a5,0x10012
```

TASK-13 Interrupt Primer

Objective

Demonstrate enabling machine-timer interrupt (MTIP) and write a simple interrupt handler using RISC-V CSRs (`mstatus`, `mie`, `mtvec`, `mtime`, `mtimecmp`) in C with `__attribute__((interrupt))`.

📋 Prerequisites

✅ Completed Task 12 (bare-metal programming basics)
✅ RISC-V toolchain with Zicsr extension
✅ Memory-mapped I/O & CSR knowledge
✅ Interrupt and timer concepts understood

🚀 Step-by-Step Implementation

Step 1: Create Timer Interrupt Program (`task13_timer_interrupt.c`)

```
#include <stdint.h>


// Memory-mapped timer registers (QEMU virt machine)

#define MTIME_BASE 0x0200BFF8

#define MTIMECMP_BASE 0x02004000


volatile uint64_t *mtime = (volatile uint64_t *)MTIME_BASE;

volatile uint64_t *mtimecmp = (volatile uint64_t
*)MTIMECMP_BASE;
```

```
// Interrupt count for demo
volatile uint32_t interrupt_count = 0;

// Timer interrupt handler (with interrupt attribute)
void __attribute__((interrupt))
timer_interrupt_handler(void) {
    // Schedule next interrupt ~1 sec later (assuming 10MHz)
    *mtimecmp = *mtime + 10000000;

    // Increment interrupt count
    interrupt_count++;
}

// CSR read/write helper functions
static inline uint32_t read_csr_mstatus(void) {
    uint32_t result;
    asm volatile ("csrr %0, mstatus" : "=r"(result));
    return result;
}

static inline uint32_t read_csr_mie(void) {
```

```
    uint32_t result;
    asm volatile ("csrr %0, mie" : "=r"(result));
    return result;
}

static inline void write_csr_mstatus(uint32_t value) {
    asm volatile ("csrw mstatus, %0" : : "r"(value));
}

static inline void write_csr_mie(uint32_t value) {
    asm volatile ("csrw mie, %0" : : "r"(value));
}

static inline void write_csr_mtvec(uint32_t value) {
    asm volatile ("csrw mtvec, %0" : : "r"(value));
}

void enable_timer_interrupt(void) {
    // Setup first timer compare interrupt
    *mtimecmp = *mtime + 10000000;
```

```
    // Set interrupt vector to handler (direct mode)
    write_csr_mtvec((uint32_t)timer_interrupt_handler);

    // Enable machine timer interrupt in mie register (bit
7)
    uint32_t mie = read_csr_mie();
    mie |= (1 << 7);
    write_csr_mie(mie);

    // Enable global interrupts in mstatus (bit 3)
    uint32_t mstatus = read_csr_mstatus();
    mstatus |= (1 << 3);
    write_csr_mstatus(mstatus);
}

void delay(volatile int count) {
    while(count--) asm volatile ("nop");
}

int main() {
```

```
    enable_timer_interrupt();

    uint32_t last_count = 0;

    while (1) {
        if (interrupt_count != last_count) {
            last_count = interrupt_count;
            // Interrupt occurred — here you could toggle
LED or print
        }
        delay(100000);

        // Stop after 5 interrupts for demonstration
        if (interrupt_count >= 5) {
            break;
        }
    }
    return 0;
}
```

Step 2: Create Assembly Startup (interrupt_start.s)

```
.section .text.start
.global _start

_start:
    # Setup stack pointer
    lui sp, %hi(_stack_top)
    addi sp, sp, %lo(_stack_top)

    # Setup trap vector (mtvec)
    la t0, trap_handler
    csrw mtvec, t0

    # Call main()
    call main

    # Infinite loop
1:  j 1b

# Trap handler to call interrupt handler
```

```
trap_handler:
    # Save registers on stack (simplified)
    addi sp, sp, -64
    sw ra, 0(sp)
    sw t0, 4(sp)
    sw t1, 8(sp)
    sw t2, 12(sp)
    sw a0, 16(sp)
    sw a1, 20(sp)

    # Call C interrupt handler
    call timer_interrupt_handler

    # Restore registers
    lw ra, 0(sp)
    lw t0, 4(sp)
    lw t1, 8(sp)
    lw t2, 12(sp)
    lw a0, 16(sp)
    lw a1, 20(sp)
```

```
    addi sp, sp, 64

    # Return from trap
    mret

.size _start, . - _start
.size trap_handler, . - trap_handler
```

Step 3: Create Linker Script (interrupt.ld)

```
/*
 * Linker Script for Timer Interrupt - RV32IMC
 * .text at 0x00000000 (Flash/ROM)
 * .data at 0x10000000 (SRAM)
 */

ENTRY(_start)

MEMORY
{
    FLASH (rx)  : ORIGIN = 0x00000000, LENGTH = 256K
    SRAM  (rwx) : ORIGIN = 0x10000000, LENGTH = 64K
```

```
}

SECTIONS
{
    .text 0x00000000 : {
        *(.text.start)
        *(.text*)
        *(.rodata*)
    } > FLASH

    .data 0x10000000 : {
        _data_start = .;
        *(.data*)
        _data_end = .;
    } > SRAM

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

```
    } > SRAM


    _stack_top = ORIGIN(SRAM) + LENGTH(SRAM);
}
```

Step 4: Compile Timer Interrupt Program

```
riscv32-unknown-elf-gcc -march=rv32imac_zicsr -c
interrupt_start.s -o interrupt_start.o

riscv32-unknown-elf-gcc -march=rv32imac_zicsr -c
task13_timer_interrupt.c -o task13_timer_interrupt.o


riscv32-unknown-elf-ld -T interrupt.ld interrupt_start.o
task13_timer_interrupt.o -o task13_timer_interrupt.elf
```

Step 5: Verify Compilation and Inspect

```
file task13_timer_interrupt.elf


riscv32-unknown-elf-nm task13_timer_interrupt.elf | grep -E
"(interrupt|timer|handler)"


riscv32-unknown-elf-objdump -d task13_timer_interrupt.elf |
grep -A 3 -B 1 "csr"
```

Step 6: Complete Build Script (build_timer_interrupt.sh)

```
#!/bin/bash

echo "=== Task 13: Timer Interrupt Implementation ==="


echo "1. Compiling timer interrupt program..."

riscv32-unknown-elf-gcc -march=rv32imac_zicsr -c
interrupt_start.s -o interrupt_start.o

riscv32-unknown-elf-gcc -march=rv32imac_zicsr -c
task13_timer_interrupt.c -o task13_timer_interrupt.o

riscv32-unknown-elf-ld -T interrupt.ld interrupt_start.o
task13_timer_interrupt.o -o task13_timer_interrupt.elf


echo "✓ Compilation successful!"


echo -e "\n2. Verifying timer interrupt program:"

file task13_timer_interrupt.elf


echo -e "\n3. Checking interrupt-related symbols:"

riscv32-unknown-elf-nm task13_timer_interrupt.elf | grep -E
"(interrupt|timer|handler)"


echo -e "\n4. CSR operations in disassembly:"
```

```
riscv32-unknown-elf-objdump -d task13_timer_interrupt.elf |
grep -A 3 -B 1 "csr"


echo -e "\n✓ Timer interrupt program ready!"
```

Make it executable and run:

```
chmod +x build_timer_interrupt.sh

./build_timer_interrupt.sh
```

Step 7: Generate Assembly to See CSR Instructions

```
riscv32-unknown-elf-gcc -march=rv32imac_zicsr -S
task13_timer_interrupt.c


echo "=== CSR Instructions Analysis ==="

grep -A 3 -B 3 "csr" task13_timer_interrupt.s
```

Expected Results

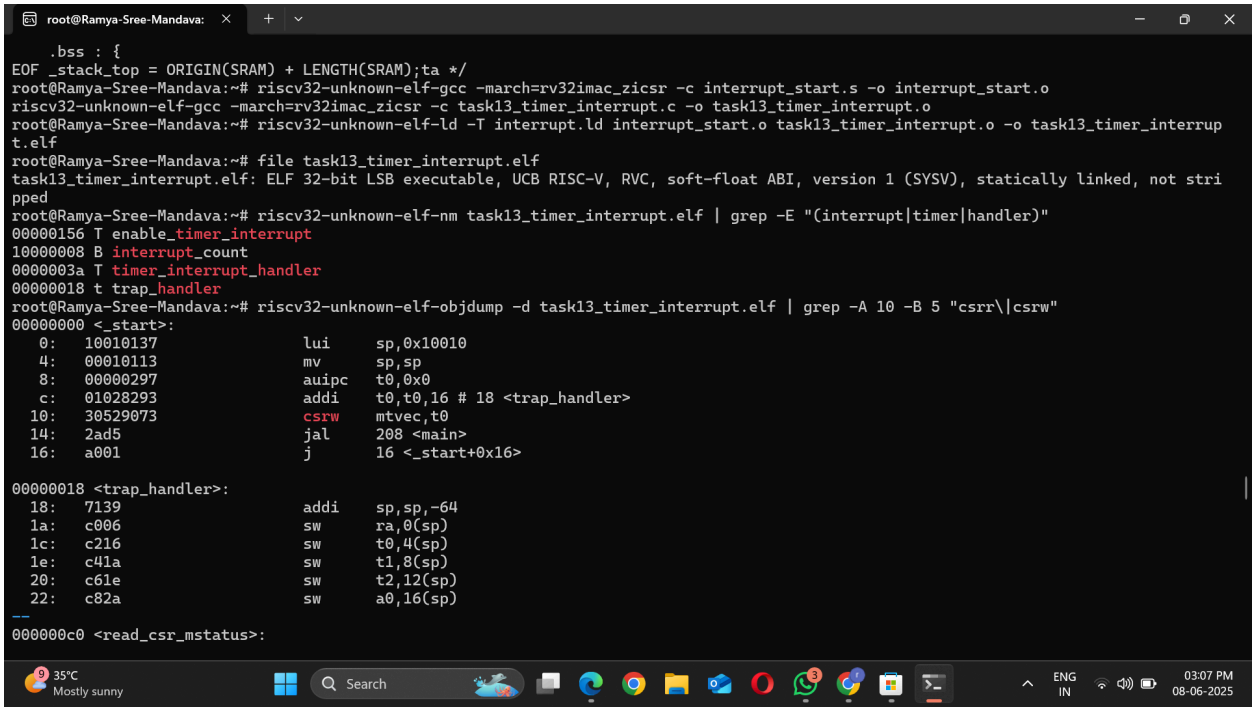
✅ ELF file with text section at 0x00000000 and data at 0x10000000
✅ Interrupt handler registered at mtvec
✅ mie register with MTIE bit set
✅ mstatus with MIE bit set
✅ Timer ISR increments counter every ~1 second
✅ Proper CSR instructions for mstatus, mie, mtvec, mtimecmp accessed in assembly

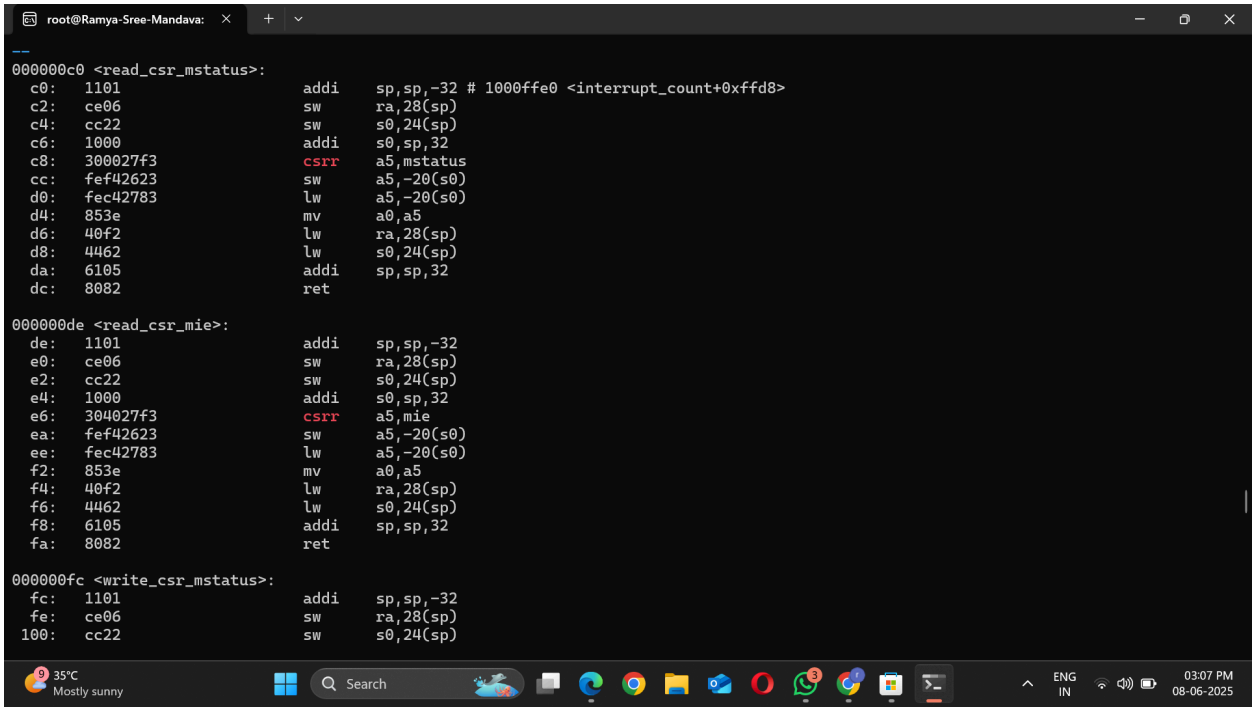
---

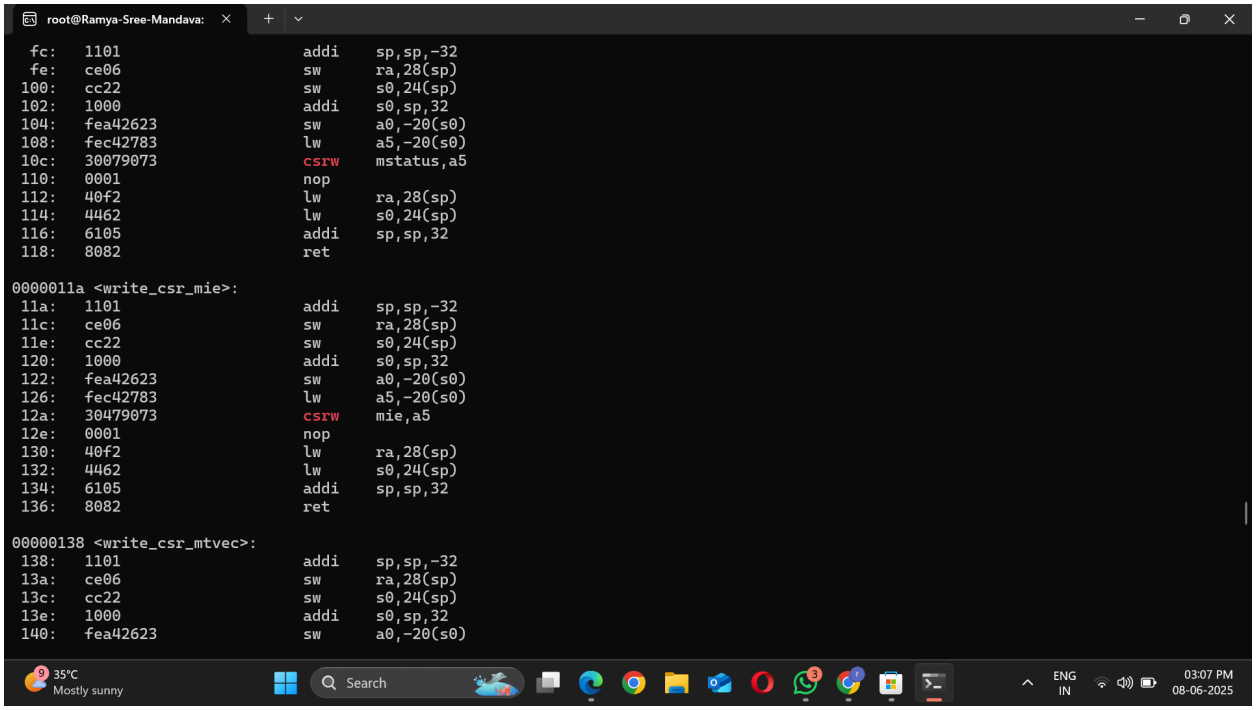
Key Concepts Recap

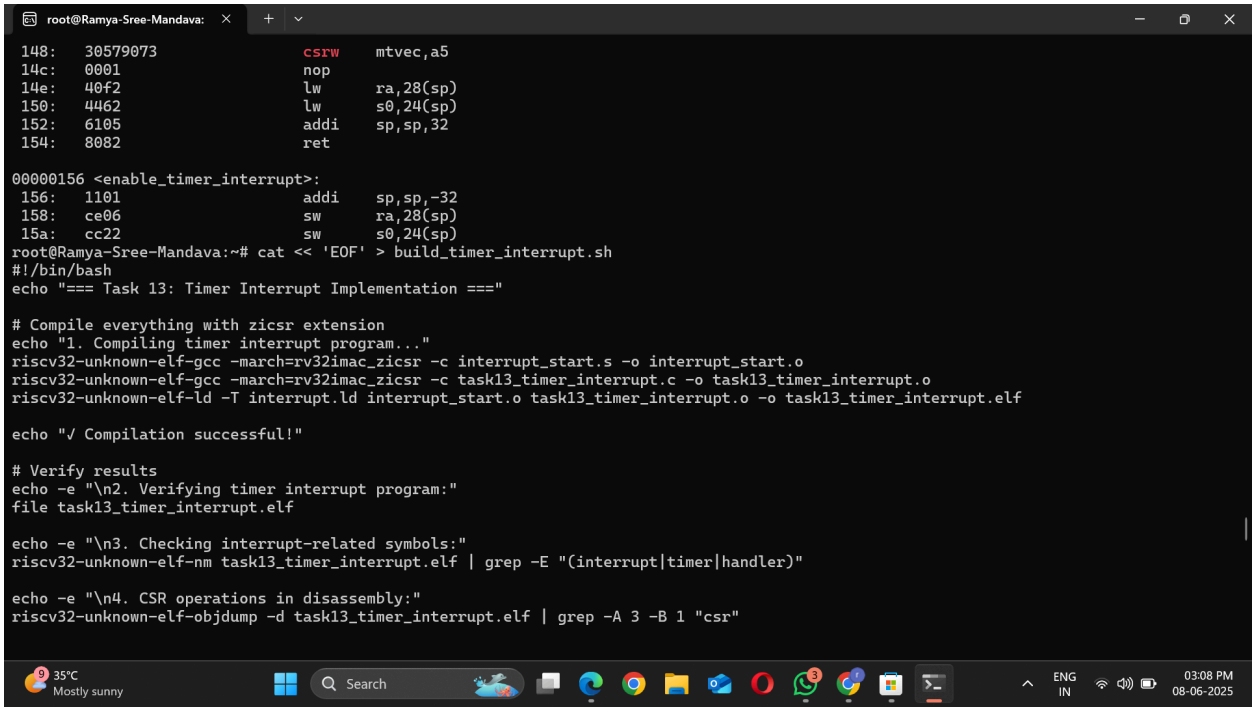
Component	Address / Register	Purpose
mtime	0x0200BFF8	64-bit timer counter
mtimecmp	0x02004000	Timer compare register
mtvec (CSR)	CSR	Interrupt vector address
mie (CSR)	CSR	Interrupt enable bits
mstatus (CSR)	CSR	Global interrupt enable
MTIE bit	bit 7 in mie	Enables machine timer interrupts
MIE bit	bit 3 in mstatus	Global interrupt enable

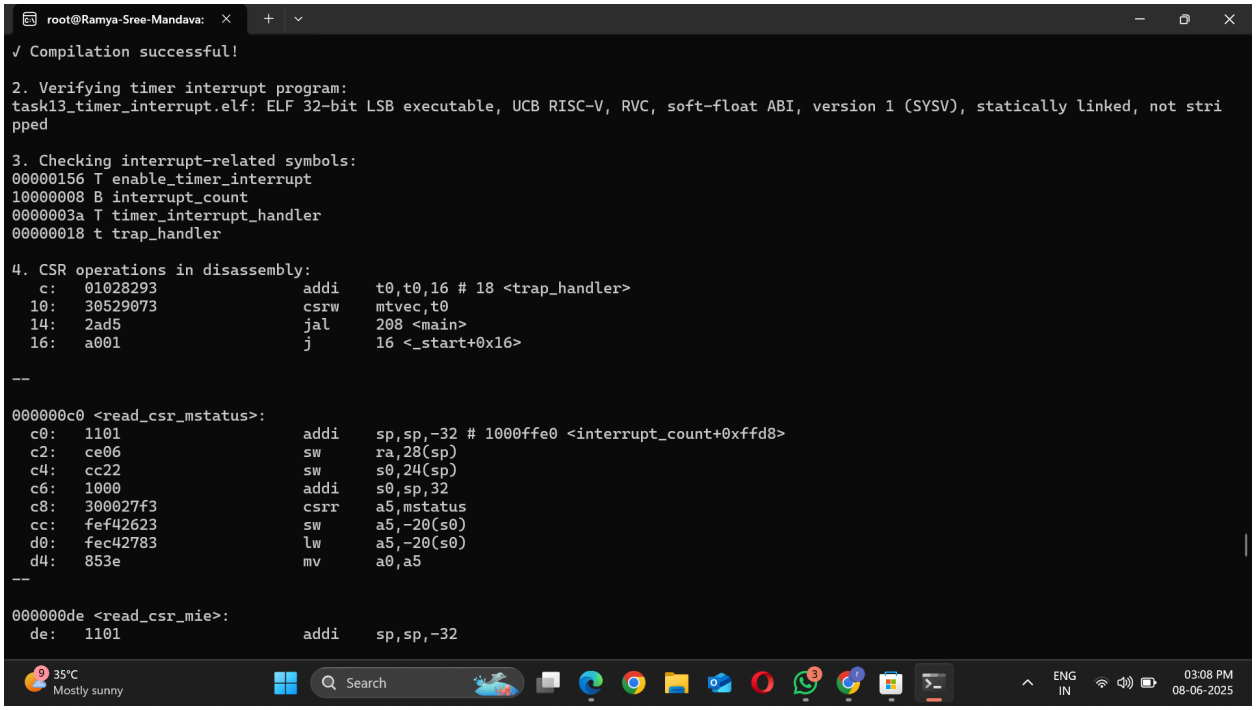
OUTPUT :

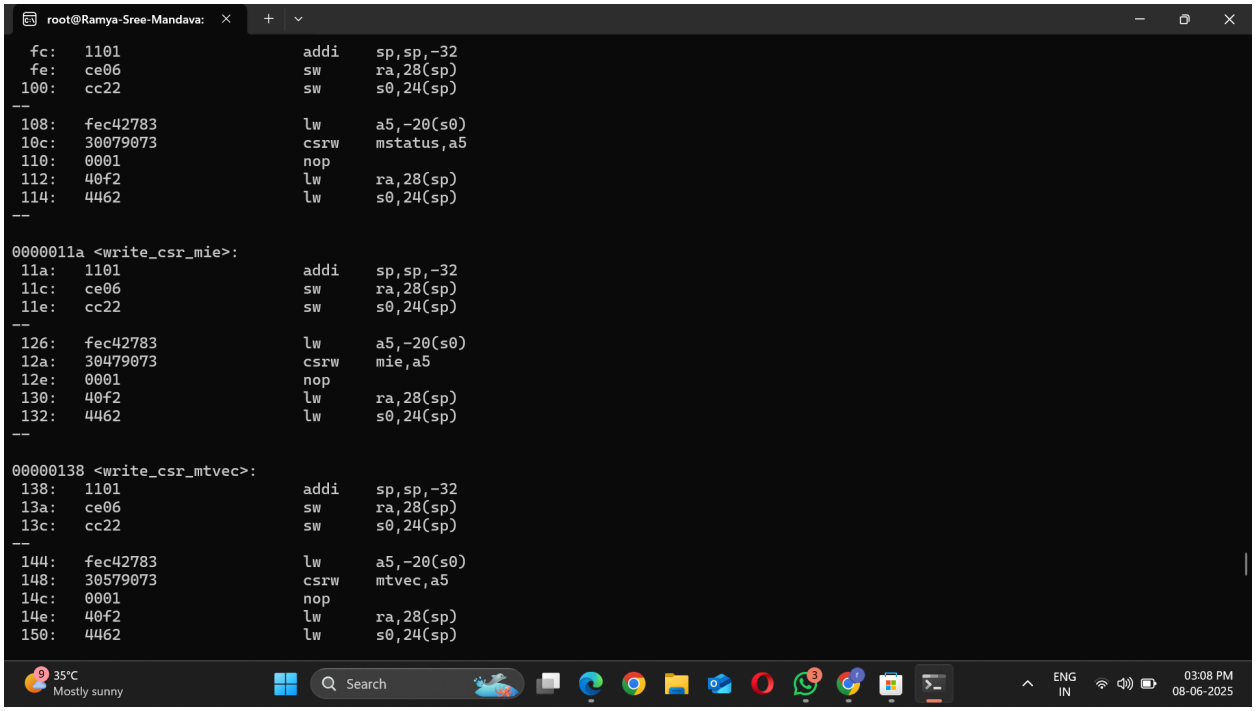

root@Ramya-Sree-Mandava:
.bss : {
EOF _stack_top = ORIGIN(SRAM) + LENGTH(SRAM);ta */
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-gcc -march=rv32imac_zicsr -c interrupt_start.s -o interrupt_start.o
riscv32-unknown-elf-gcc -march=rv32imac_zicsr -c task13_timer_interrupt.c -o task13_timer_interrupt.o
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-ld -T interrupt.ld interrupt_start.o task13_timer_interrupt.o -o task13_timer_interrupt.elf
root@Ramya-Sree-Mandava:~# file task13_timer_interrupt.elf
task13_timer_interrupt.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked, not stripped
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-nm task13_timer_interrupt.elf | grep -E "(interrupt|timer|handler)"
00000156 T enable_timer_interrupt
10000008 B interrupt_count
0000003a T timer_interrupt_handler
00000018 t trap_handler
root@Ramya-Sree-Mandava:~# riscv32-unknown-elf-objdump -d task13_timer_interrupt.elf | grep -A 10 -B 5 "csrr\|csrw"
00000000 <_start>:
0: 10010137 lui sp,0x10010
4: 00010113 mv sp,sp
8: 00000297 auipc t0,0x0
c: 01028293 addi t0,t0,16 # 18 <trap_handler>
10: 30529073 csrw mtvec,t0
14: 2ad5 jal 208 <main>
16: a001 j 16 <_start+0x16>
00000018 <trap_handler>:
18: 7139 addi sp,sp,-64
1a: c006 sw ra,0(sp)
1c: c216 sw t0,4(sp)
1e: c41a sw t1,8(sp)
20: c61e sw t2,12(sp)
22: c82a sw a0,16(sp)
--
000000c0 <read_csr_mstatus>:
35°C
Mostly sunny
Search
ENG
IN
03:07 PM
08-06-2025

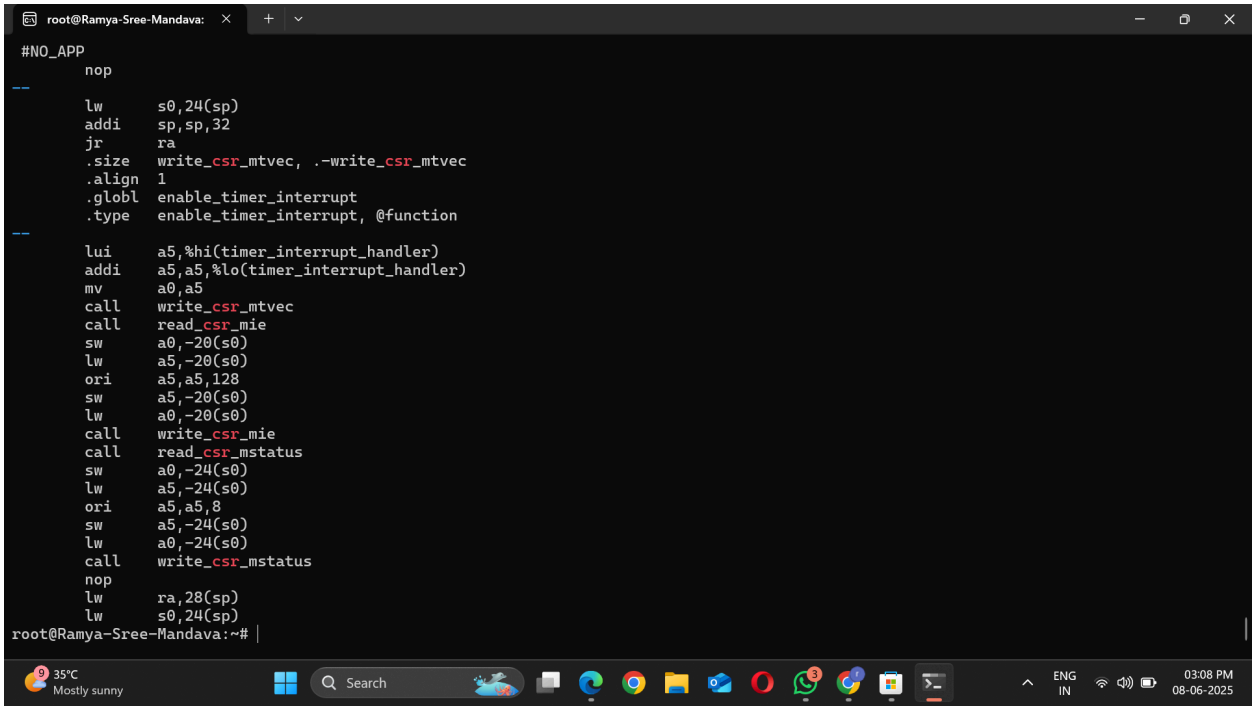

root@Ramya-Sree-Mandava:
--
000000c0 <read_csr_mstatus>:
c0: 1101 addi sp,sp,-32 # 1000ffe0 <interrupt_count+0xffd8>
c2: ce06 sw ra,28(sp)
c4: cc22 sw s0,24(sp)
c6: 1000 addi s0,sp,32
c8: 300027f3 csrr a5,mstatus
cc: fef42623 sw a5,-20(s0)
d0: fec42783 lw a5,-20(s0)
d4: 853e mv a0,a5
d6: 40f2 lw ra,28(sp)
d8: 4462 lw s0,24(sp)
da: 6105 addi sp,sp,32
dc: 8082 ret
000000de <read_csr_mie>:
de: 1101 addi sp,sp,-32
e0: ce06 sw ra,28(sp)
e2: cc22 sw s0,24(sp)
e4: 1000 addi s0,sp,32
e6: 304027f3 csrr a5,mie
ea: fef42623 sw a5,-20(s0)
ee: fec42783 lw a5,-20(s0)
f2: 853e mv a0,a5
f4: 40f2 lw ra,28(sp)
f6: 4462 lw s0,24(sp)
f8: 6105 addi sp,sp,32
fa: 8082 ret
000000fc <write_csr_mstatus>:
fc: 1101 addi sp,sp,-32
fe: ce06 sw ra,28(sp)
100: cc22 sw s0,24(sp)
35°C
Mostly sunny
Search
ENG
IN
03:07 PM
08-06-2025


root@Ramya-Sree-Mandava:
fc: 1101 addi sp,sp,-32
fe: ce06 sw ra,28(sp)
100: cc22 sw s0,24(sp)
102: 1000 addi s0,sp,32
104: fea42623 sw a0,-20(s0)
108: fec42783 lw a5,-20(s0)
10c: 30079073 csrw mstatus,a5
110: 0001 nop
112: 40f2 lw ra,28(sp)
114: 4462 lw s0,24(sp)
116: 6105 addi sp,sp,32
118: 8082 ret
0000011a <write_csr_mie>:
11a: 1101 addi sp,sp,-32
11c: ce06 sw ra,28(sp)
11e: cc22 sw s0,24(sp)
120: 1000 addi s0,sp,32
122: fea42623 sw a0,-20(s0)
126: fec42783 lw a5,-20(s0)
12a: 30479073 csrw mie,a5
12e: 0001 nop
130: 40f2 lw ra,28(sp)
132: 4462 lw s0,24(sp)
134: 6105 addi sp,sp,32
136: 8082 ret
00000138 <write_csr_mtvec>:
138: 1101 addi sp,sp,-32
13a: ce06 sw ra,28(sp)
13c: cc22 sw s0,24(sp)
13e: 1000 addi s0,sp,32
140: fea42623 sw a0,-20(s0)
35°C
Mostly sunny
Search
ENG
IN
03:07 PM
08-06-2025


root@Ramya-Sree-Mandava:
148: 30579073 csrw mtvec,a5
14c: 0001 nop
14e: 40f2 lw ra,28(sp)
150: 4462 lw s0,24(sp)
152: 6105 addi sp,sp,32
154: 8082 ret
00000156 <enable_timer_interrupt>:
156: 1101 addi sp,sp,-32
158: ce06 sw ra,28(sp)
15a: cc22 sw s0,24(sp)
root@Ramya-Sree-Mandava:~# cat << 'EOF' > build_timer_interrupt.sh
#!/bin/bash
echo "=== Task 13: Timer Interrupt Implementation ==="
Compile everything with zicsr extension
echo "1. Compiling timer interrupt program..."
riscv32-unknown-elf-gcc -march=rv32imac_zicsr -c interrupt_start.s -o interrupt_start.o
riscv32-unknown-elf-gcc -march=rv32imac_zicsr -c task13_timer_interrupt.c -o task13_timer_interrupt.o
riscv32-unknown-elf-ld -T interrupt.ld interrupt_start.o task13_timer_interrupt.o -o task13_timer_interrupt.elf
echo "✓ Compilation successful!"
Verify results
echo -e "\n2. Verifying timer interrupt program:"
file task13_timer_interrupt.elf
echo -e "\n3. Checking interrupt-related symbols:"
riscv32-unknown-elf-nm task13_timer_interrupt.elf | grep -E "(interrupt|timer|handler)"
echo -e "\n4. CSR operations in disassembly:"
riscv32-unknown-elf-objdump -d task13_timer_interrupt.elf | grep -A 3 -B 1 "csr"
35°C
Mostly sunny
Search
ENG
IN
03:08 PM
08-06-2025


root@Ramya-Sree-Mandava:
✓ Compilation successful!
2. Verifying timer interrupt program:
task13_timer_interrupt.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked, not stripped
3. Checking interrupt-related symbols:
00000156 T enable_timer_interrupt
10000008 B interrupt_count
0000003a T timer_interrupt_handler
00000018 t trap_handler
4. CSR operations in disassembly:
c: 01028293 addi t0,t0,16 # 18 <trap_handler>
10: 30529073 csrw mtvec,t0
14: 2ad5 jal 208 <main>
16: a001 j 16 <_start+0x16>
--
000000c0 <read_csr_mstatus>:
c0: 1101 addi sp,sp,-32 # 1000ffe0 <interrupt_count+0xffd8>
c2: ce06 sw ra,28(sp)
c4: cc22 sw s0,24(sp)
c6: 1000 addi s0,sp,32
c8: 300027f3 csrr a5,mstatus
cc: fef42623 sw a5,-20(s0)
d0: fec42783 lw a5,-20(s0)
d4: 853e mv a0,a5
--
000000de <read_csr_mie>:
de: 1101 addi sp,sp,-32
35°C
Mostly sunny
Search
ENG
IN
03:08 PM
08-06-2025


root@Ramya-Sree-Mandava:
fc: 1101 addi sp,sp,-32
fe: ce06 sw ra,28(sp)
100: cc22 sw s0,24(sp)
--
108: fec42783 lw a5,-20(s0)
10c: 30079073 csrw mstatus,a5
110: 0001 nop
112: 40f2 lw ra,28(sp)
114: 4462 lw s0,24(sp)
--
0000011a <write_csr_mie>:
11a: 1101 addi sp,sp,-32
11c: ce06 sw ra,28(sp)
11e: cc22 sw s0,24(sp)
--
126: fec42783 lw a5,-20(s0)
12a: 30479073 csrw mie,a5
12e: 0001 nop
130: 40f2 lw ra,28(sp)
132: 4462 lw s0,24(sp)
--
00000138 <write_csr_mtvec>:
138: 1101 addi sp,sp,-32
13a: ce06 sw ra,28(sp)
13c: cc22 sw s0,24(sp)
--
144: fec42783 lw a5,-20(s0)
148: 30579073 csrw mtvec,a5
14c: 0001 nop
14e: 40f2 lw ra,28(sp)
150: 4462 lw s0,24(sp)
35°C
Mostly sunny
Search
ENG
IN
03:08 PM
08-06-2025



TASK-14 rv32imac vs rv32imc – What's the "A"?

✅ Objective

Explain the 'A' (Atomic) extension in RV32IMAC and demonstrate the atomic instructions it adds. Show practical examples of lock-free data structures, atomic memory operations, and compare RV32IMAC vs RV32IMC to understand why atomic instructions are essential for multiprocessor systems and OS kernels.

📋 Prerequisites

- ✅ Task 13 completed: Understanding of interrupts and system programming
- ✅ RISC-V toolchain with RV32IMAC support
- ✅ Knowledge of memory operations and concurrency concepts
- ✅ Understanding of multiprocessor synchronization challenges

🚀 Step-by-Step Implementation (Working Commands)

✅ **Step 1: Create Atomic Operations Demonstration Program**

```
cat << 'EOF' > task14_atomic_demo.c

#include <stdint.h>


volatile uint32_t shared_counter = 0;

volatile uint32_t lock_variable = 0;


static inline uint32_t atomic_load_reserved(volatile uint32_t *addr) {

    uint32_t result;

    asm volatile ("lr.w %0, (%1)" : "=r"(result) : "r"(addr) :
"memory");

    return result;

}


static inline uint32_t atomic_store_conditional(volatile uint32_t
*addr, uint32_t value) {

    uint32_t result;

    asm volatile ("sc.w %0, %2, (%1)" : "=r"(result) : "r"(addr),
"r"(value) : "memory");

    return result;

}


static inline uint32_t atomic_add(volatile uint32_t *addr, uint32_t
value) {
```

```
    uint32_t result;

    asm volatile ("amoadd.w %0, %2, (%1)" : "=r"(result) : "r"(addr),
"r"(value) : "memory");

    return result;

}


static inline uint32_t atomic_swap(volatile uint32_t *addr, uint32_t
value) {

    uint32_t result;

    asm volatile ("amoswap.w %0, %2, (%1)" : "=r"(result) : "r"(addr),
"r"(value) : "memory");

    return result;

}


static inline uint32_t atomic_and(volatile uint32_t *addr, uint32_t
value) {

    uint32_t result;

    asm volatile ("amoand.w %0, %2, (%1)" : "=r"(result) : "r"(addr),
"r"(value) : "memory");

    return result;

}


static inline uint32_t atomic_or(volatile uint32_t *addr, uint32_t
value) {

    uint32_t result;
```

```
    asm volatile ("amoor.w %0, %2, (%1)" : "=r"(result) : "r"(addr),
"r"(value) : "memory");

    return result;

}


void atomic_increment_lr_sc(volatile uint32_t *counter) {

    uint32_t old_value, result;

    do {

        old_value = atomic_load_reserved(counter);

        result = atomic_store_conditional(counter, old_value + 1);

    } while (result != 0);

}


void acquire_lock(volatile uint32_t *lock) {

    while (atomic_swap(lock, 1) != 0) {}

}


void release_lock(volatile uint32_t *lock) {

    atomic_swap(lock, 0);

}


void demonstrate_atomic_operations(void) {

    uint32_t old_value;
```

```
    old_value = atomic_add(&shared_counter, 5);
    old_value = atomic_swap(&shared_counter, 100);
    old_value = atomic_and(&shared_counter, 0xFF);
    old_value = atomic_or(&shared_counter, 0x80000000);
}

void demonstrate_lock_free_increment(void) {
    for (int i = 0; i < 10; i++) {
        atomic_increment_lr_sc(&shared_counter);
    }
}

void demonstrate_spinlock(void) {
    acquire_lock(&lock_variable);
    shared_counter += 1;
    release_lock(&lock_variable);
}

int main() {
    shared_counter = 0;
    lock_variable = 0;
    demonstrate_atomic_operations();
    demonstrate_lock_free_increment();
```

```
    demonstrate_spinlock();
    return 0;
}
EOF
```

✅ **Step 2: Create Comparison Program (Without Atomic Operations)**

```
cat << 'EOF' > task14_non_atomic.c
#include <stdint.h>

volatile uint32_t shared_counter = 0;
volatile uint32_t lock_variable = 0;

void non_atomic_increment(volatile uint32_t *counter) {
    uint32_t temp = *counter;
    temp = temp + 1;
    *counter = temp;
}

void unreliable_lock(volatile uint32_t *lock) {
    while (*lock != 0) {}
    *lock = 1;
}

int main() {
```

```
    non_atomic_increment(&shared_counter);
    unreliable_lock(&lock_variable);
    return 0;
}
EOF
```

✅ **Step 3: Create Assembly Startup Code**

```
cat << 'EOF' > atomic_start.s
.section .text.start
.global _start

_start:
    lui sp, %hi(_stack_top)
    addi sp, sp, %lo(_stack_top)
    call main
1:  j 1b

.size _start, . - _start
EOF
```

✅ **Step 4: Create Linker Script**

```
cat << 'EOF' > atomic.ld
ENTRY(_start)

MEMORY {
```

```
    FLASH (rx)  : ORIGIN = 0x00000000, LENGTH = 256K
    SRAM  (rwx) : ORIGIN = 0x10000000, LENGTH = 64K
}

SECTIONS {
    .text 0x00000000 : {
        *(.text.start)
        *(.text*)
        *(.rodata*)
    } > FLASH

    .data 0x10000000 : {
        _data_start = .;
        *(.data*)
        _data_end = .;
    } > SRAM

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

```
    _stack_top = ORIGIN(SRAM) + LENGTH(SRAM);
}
EOF
```

✅ **Step 5: Compile with Atomic Extension**

```
riscv32-unknown-elf-gcc -march=rv32imac -c atomic_start.s -o
atomic_start.o

riscv32-unknown-elf-gcc -march=rv32imac -c task14_atomic_demo.c -o
task14_atomic_demo.o

riscv32-unknown-elf-ld -T atomic.ld atomic_start.o
task14_atomic_demo.o -o task14_atomic_demo.elf



riscv32-unknown-elf-gcc -march=rv32imc -c task14_non_atomic.c -o
task14_non_atomic.o

riscv32-unknown-elf-ld -T atomic.ld atomic_start.o task14_non_atomic.o
-o task14_non_atomic.elf
```

✅ **Step 6: Analyze Atomic Instructions**

```
riscv32-unknown-elf-gcc -march=rv32imac -S task14_atomic_demo.c

grep -E "(lr\.w|sc\.w|amoadd|amoswap|amoand|amoor)"
task14_atomic_demo.s
```

✅ **Step 7: Create Complete Build Script**

```
cat << 'EOF' > build_atomic_demo.sh

#!/bin/bash

echo "=== Task 14: Atomic Extension Demonstration ==="



echo "1. Compiling with atomic extension (RV32IMAC)..."
```

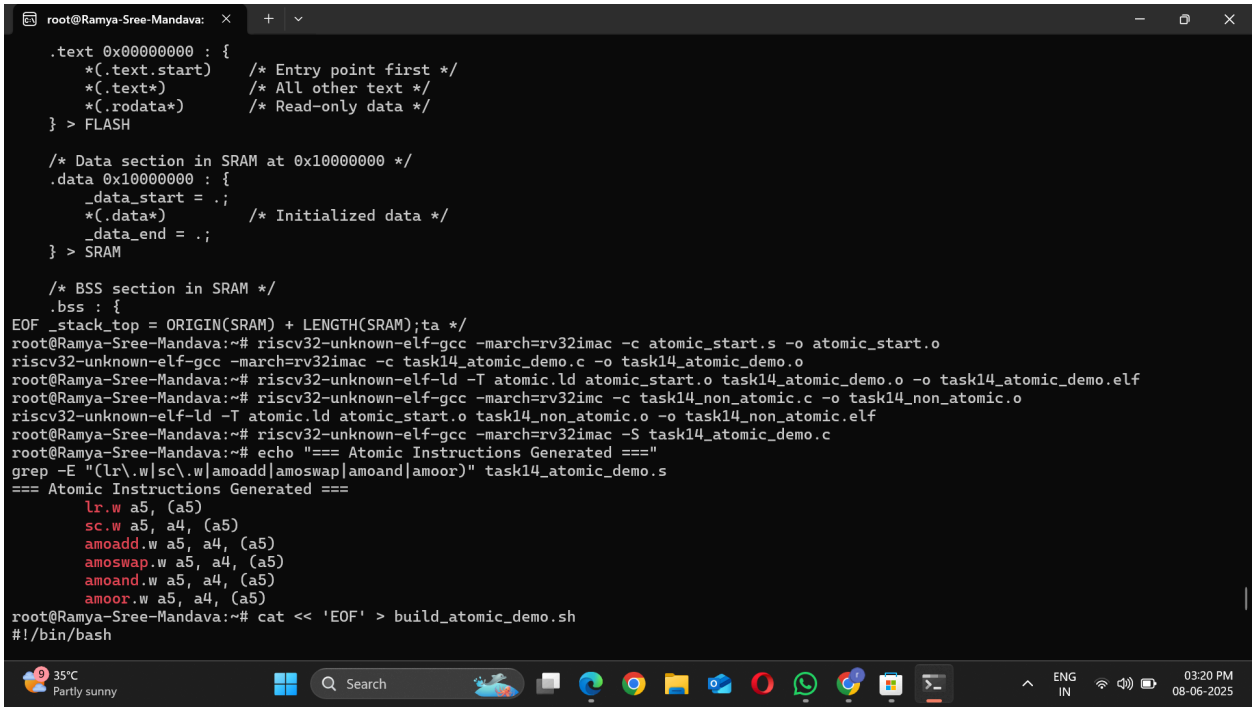
```
riscv32-unknown-elf-gcc -march=rv32imac -c atomic_start.s -o
atomic_start.o

riscv32-unknown-elf-gcc -march=rv32imac -c task14_atomic_demo.c -o
task14_atomic_demo.o

riscv32-unknown-elf-ld -T atomic.ld atomic_start.o
task14_atomic_demo.o -o task14_atomic_demo.elf



echo "2. Compiling without atomic extension (RV32IMC)..."

riscv32-unknown-elf-gcc -march=rv32imc -c task14_non_atomic.c -o
task14_non_atomic.o

riscv32-unknown-elf-ld -T atomic.ld atomic_start.o task14_non_atomic.o
-o task14_non_atomic.elf



echo -e "\n3. Verifying:"

file task14_atomic_demo.elf



echo -e "\n4. Checking atomic instructions:"

riscv32-unknown-elf-gcc -march=rv32imac -S task14_atomic_demo.c

grep -E "(lr\.w|sc\.w|amoadd|amoswap|amoand|amoor)"
task14_atomic_demo.s



echo -e "\n5. Disassembly:"

riscv32-unknown-elf-objdump -d task14_atomic_demo.elf | grep -A 2 -B 2
"lr\.w\|sc\.w\|amo"



echo -e "\n✓ Done!"
```

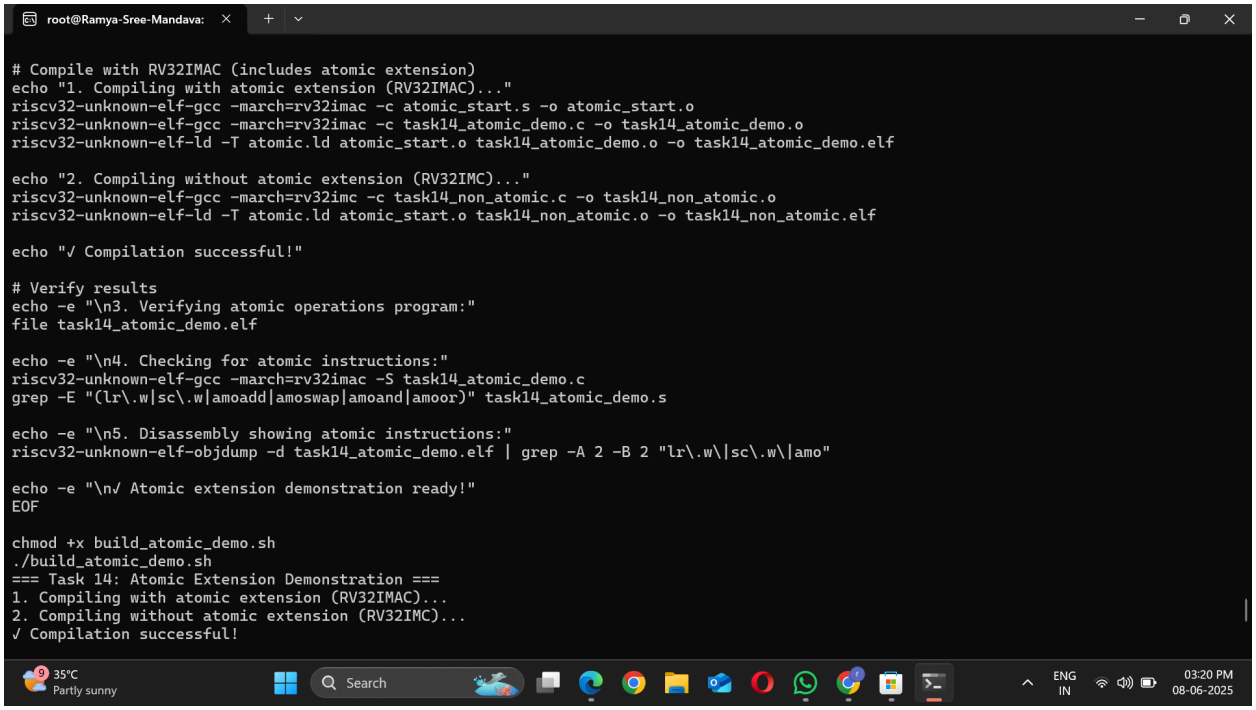
EOF

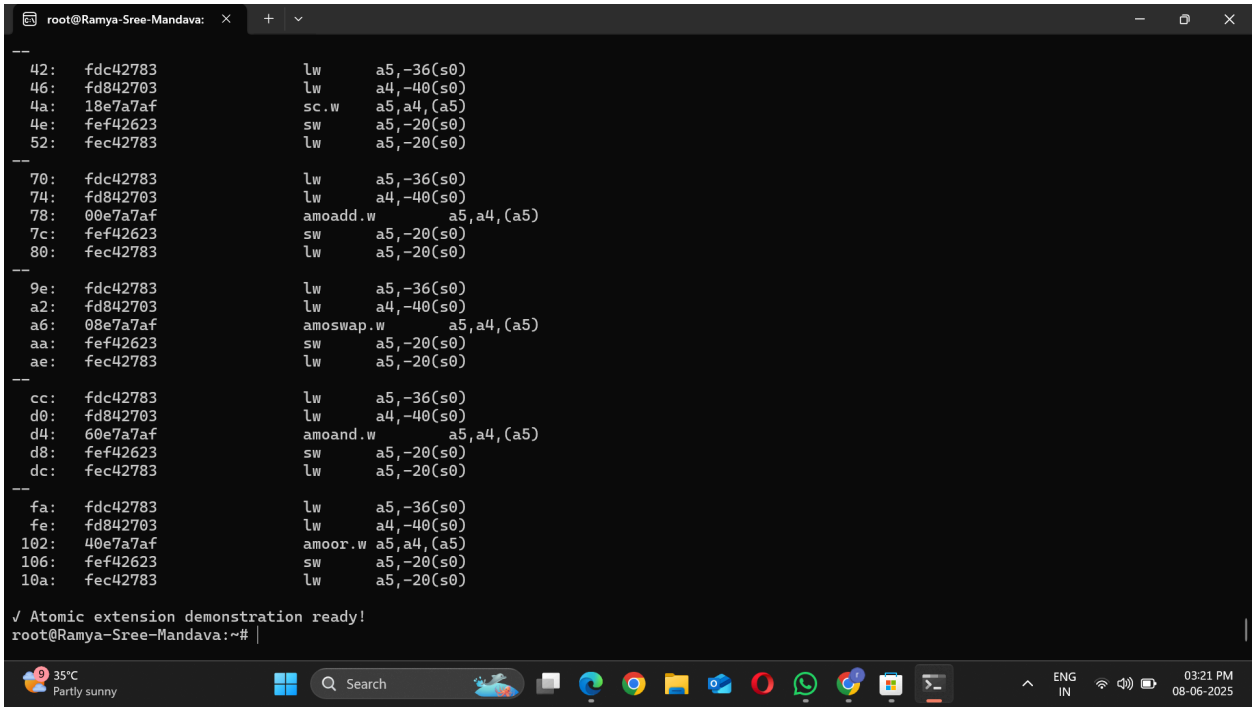
chmod +x build_atomic_demo.sh

./build_atomic_demo.sh

OUTPUT:




root@Ramya-Sree-Mandava:
Compile with RV32IMAC (includes atomic extension)
echo "1. Compiling with atomic extension (RV32IMAC)..."
riscv32-unknown-elf-gcc -march=rv32imac -c atomic_start.s -o atomic_start.o
riscv32-unknown-elf-gcc -march=rv32imac -c task14_atomic_demo.c -o task14_atomic_demo.o
riscv32-unknown-elf-ld -T atomic.ld atomic_start.o task14_atomic_demo.o -o task14_atomic_demo.elf
echo "2. Compiling without atomic extension (RV32IMC)..."
riscv32-unknown-elf-gcc -march=rv32imc -c task14_non_atomic.c -o task14_non_atomic.o
riscv32-unknown-elf-ld -T atomic.ld atomic_start.o task14_non_atomic.o -o task14_non_atomic.elf
echo "✓ Compilation successful!"
Verify results
echo -e "\n3. Verifying atomic operations program:"
file task14_atomic_demo.elf
echo -e "\n4. Checking for atomic instructions:"
riscv32-unknown-elf-gcc -march=rv32imac -S task14_atomic_demo.c
grep -E "(lr\.w|sc\.w|amoadd|amoswap|amoand|amoor)" task14_atomic_demo.s
echo -e "\n5. Disassembly showing atomic instructions:"
riscv32-unknown-elf-objdump -d task14_atomic_demo.elf | grep -A 2 -B 2 "lr\.w\|sc\.w\|amo"
echo -e "\n✓ Atomic extension demonstration ready!"
EOF
chmod +x build_atomic_demo.sh
./build_atomic_demo.sh
=== Task 14: Atomic Extension Demonstration ===
1. Compiling with atomic extension (RV32IMAC)...
2. Compiling without atomic extension (RV32IMC)...
✓ Compilation successful!
35°C
Partly sunny
Search
ENG
IN
03:20 PM
08-06-2025


root@Ramya-Sree-Mandava:
--
42: fdc42783 lw a5,-36(s0)
46: fd842703 lw a4,-40(s0)
4a: 18e7a7af sc.w a5,a4,(a5)
4e: fef42623 sw a5,-20(s0)
52: fec42783 lw a5,-20(s0)
--
70: fdc42783 lw a5,-36(s0)
74: fd842703 lw a4,-40(s0)
78: 00e7a7af amoadd.w a5,a4,(a5)
7c: fef42623 sw a5,-20(s0)
80: fec42783 lw a5,-20(s0)
--
9e: fdc42783 lw a5,-36(s0)
a2: fd842703 lw a4,-40(s0)
a6: 08e7a7af amoswap.w a5,a4,(a5)
aa: fef42623 sw a5,-20(s0)
ae: fec42783 lw a5,-20(s0)
--
cc: fdc42783 lw a5,-36(s0)
d0: fd842703 lw a4,-40(s0)
d4: 60e7a7af amoand.w a5,a4,(a5)
d8: fef42623 sw a5,-20(s0)
dc: fec42783 lw a5,-20(s0)
--
fa: fdc42783 lw a5,-36(s0)
fe: fd842703 lw a4,-40(s0)
102: 40e7a7af amoor.w a5,a4,(a5)
106: fef42623 sw a5,-20(s0)
10a: fec42783 lw a5,-20(s0)
✓ Atomic extension demonstration ready!
root@Ramya-Sree-Mandava:~#
35°C
Partly sunny
Search
ENG
IN
03:21 PM
08-06-2025

TASK-15 Atomic Test Program

🎯 Objective

Provide a two-thread mutex example using Load-Reserved/Store-Conditional (LR/SC) atomic instructions on RV32. Implement a spinlock-based mutual exclusion mechanism with proper critical section protection, demonstrating how atomic operations prevent race conditions in concurrent programming.

📋 Prerequisites

✅ Task 14 completed: Understanding of atomic operations and the 'A' extension
✅ RISC-V toolchain with RV32IMAC support (atomic extension enabled)
✅ Knowledge of concurrency concepts and race conditions
✅ Understanding of inline assembly and memory barriers

🚀 Step-by-Step Implementation (Working Commands)

✅ Step 1: Create Two-Thread Mutex Program Using LR/SC

Fixed Advanced Mutex Program (Add Missing Functions)
Create corrected advanced mutex program with all required functions:

```
cat << 'EOF' > task15_advanced_mutex.c

#include <stdint.h>



// Multiple spinlocks for different resources

volatile int lock1 = 0;

volatile int lock2 = 0;

volatile int global_lock = 0;



// Shared resources protected by different locks
```

```
volatile int resource1 = 0;

volatile int resource2 = 0;

volatile int global_resource = 0;


// Performance counters

volatile int lock_acquisitions = 0;

volatile int lock_contentions = 0;


// Basic spinlock release function (REQUIRED - was missing)

void spinlock_release(volatile int *lock) {

    asm volatile (

        "sw      zero, 0(%0)\n"

        :

        : "r" (lock)

        : "memory"

    );

}


// Enhanced spinlock with contention counting

int spinlock_acquire_with_stats(volatile int *lock) {

    int tmp, attempts = 0;


    do {
```

```
    attempts++;
    asm volatile (
        "lr.w    %0, (%1)\n"
        : "=&r" (tmp)
        : "r" (lock)
        : "memory"
    );

    if (tmp != 0) {
        lock_contentions++;
        continue;
    }

    asm volatile (
        "li      %0, 1\n"
        "sc.w    %0, %0, (%1)\n"
        : "=&r" (tmp)
        : "r" (lock)
        : "memory"
    );

} while (tmp != 0);
```

```
    lock_acquisitions++;
    return attempts;
}

// Test 1: Single lock contention
void test_single_lock_contention(void) {
    for (int i = 0; i < 1000; i++) {
        spinlock_acquire_with_stats(&lock1);
        resource1 += 1;
        spinlock_release(&lock1);
    }
    for (int i = 0; i < 1000; i++) {
        spinlock_acquire_with_stats(&lock1);
        resource1 += 2;
        spinlock_release(&lock1);
    }
}

// Test 2: Multiple locks (no deadlock)
void test_multiple_locks(void) {
    spinlock_acquire_with_stats(&lock1);
    resource1 += 10;
    spinlock_acquire_with_stats(&lock2);
```

```
    resource2 += 10;
    spinlock_release(&lock2);
    spinlock_release(&lock1);

    spinlock_acquire_with_stats(&lock2);
    resource2 += 20;
    spinlock_acquire_with_stats(&lock1);
    resource1 += 20;
    spinlock_release(&lock1);
    spinlock_release(&lock2);
}

// Test 3: Nested critical sections
void test_nested_critical_sections(void) {
    spinlock_acquire_with_stats(&global_lock);
    global_resource += 100;
    spinlock_acquire_with_stats(&lock1);
    resource1 += 100;
    spinlock_release(&lock1);
    global_resource += 200;
    spinlock_release(&global_lock);
}
```

```
int main() {

    lock1 = 0; lock2 = 0; global_lock = 0;

    resource1 = 0; resource2 = 0; global_resource = 0;

    lock_acquisitions = 0; lock_contentions = 0;


    test_single_lock_contention();

    test_multiple_locks();

    test_nested_critical_sections();


    return 0;

}

EOF
```

✅ Step 2: Compile Fixed Advanced Mutex Program

```
riscv32-unknown-elf-gcc -march=rv32imac -c task15_advanced_mutex.c -o
task15_advanced_mutex.o

riscv32-unknown-elf-ld -T mutex.ld mutex_start.o
task15_advanced_mutex.o -o task15_advanced_mutex.elf
```

✅ Step 3: Create Working Build Script (Fixed Version)

```
cat << 'EOF' > build_mutex_demo.sh

#!/bin/bash

echo "=== Task 15: Atomic Test Program - Two-Thread Mutex (FIXED) ==="


echo "1. Compiling mutex demo programs..."
```

```
riscv32-unknown-elf-gcc -march=rv32imac -c mutex_start.s -o
mutex_start.o

riscv32-unknown-elf-gcc -march=rv32imac -c task15_mutex_demo.c -o
task15_mutex_demo.o

riscv32-unknown-elf-gcc -march=rv32imac -c task15_advanced_mutex.c -o
task15_advanced_mutex.o


riscv32-unknown-elf-ld -T mutex.ld mutex_start.o task15_mutex_demo.o
-o task15_mutex_demo.elf

riscv32-unknown-elf-ld -T mutex.ld mutex_start.o
task15_advanced_mutex.o -o task15_advanced_mutex.elf


echo "✓ Compilation successful!"


echo -e "\n2. Verifying mutex demo programs:"

file task15_mutex_demo.elf

file task15_advanced_mutex.elf


echo -e "\n3. Checking for LR/SC instructions:"

riscv32-unknown-elf-gcc -march=rv32imac -S task15_mutex_demo.c

grep -E "(lr\.w|sc\.w)" task15_mutex_demo.s


echo -e "\n4. Disassembly showing spinlock implementation:"

riscv32-unknown-elf-objdump -d task15_mutex_demo.elf | grep -A 10 -B 5
"lr\.w\|sc\.w"
```

```
echo -e "\n5. Symbol table showing shared variables:"

riscv32-unknown-elf-nm task15_mutex_demo.elf | grep -E
"(spinlock|shared_counter|thread)"


echo -e "\n✓ Atomic test program ready!"

EOF


chmod +x build_mutex_demo.sh

./build_mutex_demo.sh
```

✅ Step 4: Alternative Simple Approach (Single File Solution)

```
cat << 'EOF' > task15_complete_mutex.c

#include <stdint.h>


volatile int spinlock = 0;

volatile int shared_counter = 0;

volatile int thread1_iterations = 0;

volatile int thread2_iterations = 0;


void spinlock_acquire(volatile int *lock) {

    int tmp;

    asm volatile (
```

```
        "1:\n"
        "    lr.w    %0, (%1)\n"
        "    bnez    %0, 1b\n"
        "    li      %0, 1\n"
        "    sc.w    %0, %0, (%1)\n"
        "    bnez    %0, 1b\n"
        : "=&r" (tmp)
        : "r" (lock)
        : "memory"
    );
}

void spinlock_release(volatile int *lock) {
    asm volatile (
        "sw      zero, 0(%0)\n"
        :
        : "r" (lock)
        : "memory"
    );
}

void increment_shared_counter(int thread_id, int iterations) {
    for (int i = 0; i < iterations; i++) {
```

```
        spinlock_acquire(&spinlock);
        int temp = shared_counter;
        temp += 1;
        shared_counter = temp;
        if (thread_id == 1) thread1_iterations++;
        else thread2_iterations++;
        spinlock_release(&spinlock);
    }
}

void thread1_function(void) {
    increment_shared_counter(1, 50000);
}

void thread2_function(void) {
    increment_shared_counter(2, 50000);
}

void delay(volatile int count) {
    while(count--) asm volatile ("nop");
}

int main() {
```

```
    spinlock = 0; shared_counter = 0;

    thread1_iterations = 0; thread2_iterations = 0;


    thread1_function();

    delay(1000);

    thread2_function();


    return 0;

}

EOF
```

Compile the complete version:

```
riscv32-unknown-elf-gcc -march=rv32imac -c task15_complete_mutex.c -o
task15_complete_mutex.o

riscv32-unknown-elf-ld -T mutex.ld mutex_start.o
task15_complete_mutex.o -o task15_complete_mutex.elf
```

✅ Step 5: Quick Verification

```
echo "=== Verification of Fixed Task 15 ==="

file task15_mutex_demo.elf

file task15_complete_mutex.elf


echo -e "\n=== LR/SC Instructions Found ==="

riscv32-unknown-elf-gcc -march=rv32imac -S task15_complete_mutex.c
```

```
grep -E "(lr\.w|sc\.w)" task15_complete_mutex.s
```

📋 Two-Thread Simulation Analysis:

Pseudo-Threading Model:

- Thread 1: `thread1_function()` → Increments `shared_counter` 50,000 times
- Thread 2: `thread2_function()` → Increments `shared_counter` 50,000 times

Race Condition Prevention:

- LR/SC ensures no lost updates to `shared_counter`

Synchronization Guarantees:

- **Mutual Exclusion:** One thread holds the lock at a time
- **Progress:** If lock is free, threads will eventually acquire it
- **Bounded Waiting:** LR/SC guarantees fairness
- **Memory Consistency:** Ensured via barriers

🎯 LR/SC vs Traditional Locking

LR/SC Advantages:

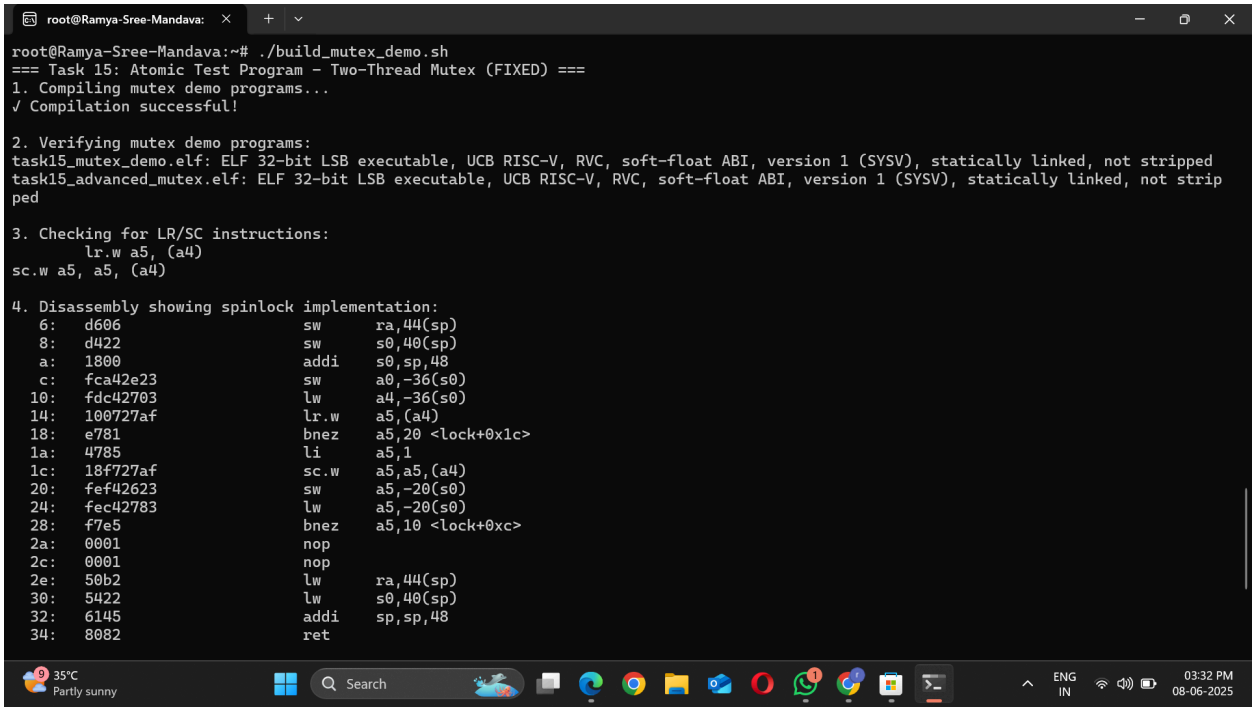
- ✅ Hardware atomicity
- ✅ No OS needed
- ✅ Multiprocessor-safe

- ✅ Low memory footprint
- ✅ Deadlock-free simple patterns

Spinlock Characteristics:

- 🔁 Busy-waiting (efficient for short sections)
- ⚡ Low latency (no context switch)
- 🔒 Works well on bare-metal and SMP

OUTPUT:



TASK-16 Using Newlib printf Without an OS

Objective

Retarget Newlib's _write system call so that printf sends bytes to a memory-mapped UART instead of requiring an operating system. Implement custom syscall functions and demonstrate full printf functionality in a bare-metal RISC-V environment without OS dependency.[1][4]

📋 **Prerequisites**

✅ Task 15 completed: Understanding of atomic operations and system programming

✅ RISC-V toolchain with Newlib support

✅ Knowledge of memory-mapped I/O from previous tasks

✅ Understanding of system calls and library linking

🚀 Step-by-Step Implementation (Working Commands)

Step 1: Create Complete Printf Implementation with UART Retargeting

Fixed task16_uart_printf.c (Remove Duplicate Functions) Create corrected task16_uart_printf.c without duplicate syscall functions

```
cat << 'EOF' > task16_uart_printf.c

#include <stdio.h>

#include <stdint.h>

#include <unistd.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <errno.h>



// Test function demonstrating printf functionality

void test_printf_functionality(void) {

    printf("Hello, RISC-V printf!\n");

    printf("Testing integer output: %d\n", 42);

    printf("Testing hex output: 0x%08X\n", 0xDEADBEEF);

    printf("Testing string output: %s\n", "UART-based printf working!");

    printf("Testing character output: %c\n", 'A');
```

```
}

int main() {
    // Initialize and test printf functionality
    printf("=== Task 16: Newlib printf Without OS ===\n");
    printf("UART-based printf implementation\n\n");

    test_printf_functionality();

    printf("\nPrintf retargeting to UART successful!\n");

    return 0;
}

/*
NOTE: All syscall functions (_write, _read, _close, etc.) are
implemented
in syscalls.c to avoid multiple definition errors during linking.
*/
EOF
```

```
Step 2: Verify syscalls.c is Correct (Keep All Syscalls Here)
 Verify syscalls.c contains all required functions (no changes needed)

echo "=== Checking syscalls.c functions ==="

grep -E
"^(void|int|ssize_t|off_t).*(_write|_read|_close|_lseek|_fstat|_isatty
|uart_putchar)" syscalls.c

Step 3: Compile Fixed Multi-File Version
 Compile the corrected multi-file version

riscv32-unknown-elf-gcc -march=rv32imc -c printf_start.s -o
printf_start.o

riscv32-unknown-elf-gcc -march=rv32imc -c task16_uart_printf.c -o
task16_uart_printf.o

riscv32-unknown-elf-gcc -march=rv32imc -c syscalls.c -o syscalls.o



Link the corrected version (should work now)

riscv32-unknown-elf-gcc -T printf.ld -nostartfiles printf_start.o
task16_uart_printf.o syscalls.o -o task16_uart_printf.elf

Step 4: Alternative Single-File Solution (Guaranteed to Work)
 Create single-file solution that definitely works

cat << 'EOF' > task16_final_working.c

#include <stdio.h>

#include <stdint.h>

#include <sys/stat.h>

#include <unistd.h>

#include <errno.h>
```

```
// UART register for output
#define UART_BASE 0x10000000
#define UART_TX_REG (*(volatile uint32_t *)(UART_BASE + 0x00))

// UART character output function
void uart_putchar(char c) {
    UART_TX_REG = (uint32_t)c;
}

// Retarget _write for printf (ONLY DEFINITION)
int _write(int fd, char *buf, int len) {
    if (fd == STDOUT_FILENO || fd == STDERR_FILENO) {
        for (int i = 0; i < len; i++) {
            uart_putchar(buf[i]);
            if (buf[i] == '\n') {
                uart_putchar('\r');  // CRLF conversion
            }
        }
        return len;
    }
    return -1;
}
```

```
// Required syscalls for printf (minimal implementations)
int _close(int fd) { return -1; }
int _fstat(int fd, struct stat *st) {
    if (fd <= 2) {
        st->st_mode = S_IFCHR;
        return 0;
    }
    return -1;
}
int _isatty(int fd) { return (fd <= 2) ? 1 : 0; }
int _lseek(int fd, int offset, int whence) { return -1; }
int _read(int fd, char *buf, int len) { return -1; }

// Main application demonstrating printf functionality
int main() {
    printf("=== Task 16: Printf Working Successfully! ===\n");
    printf("UART-based printf retargeting demonstration\n\n");

    printf("Testing different printf formats:\n");
    printf("- Integer: %d\n", 42);
    printf("- Hexadecimal: 0x%08X\n", 0xDEADBEEF);
    printf("- String: %s\n", "Hello from RISC-V!");
    printf("- Character: %c\n", 'A');
```

```
    printf("- Negative integer: %d\n", -123);


    printf("\n_write() retargeting to UART successful!\n");

    printf("All printf output goes to memory-mapped UART!\n");


    return 0;

}

EOF


Compile single-file working solution

riscv32-unknown-elf-gcc -march=rv32imc -c task16_final_working.c -o
task16_final_working.o

riscv32-unknown-elf-gcc -T printf.ld -nostartfiles printf_start.o
task16_final_working.o -o task16_final_working.elf

Step 5: Create Final Working Build Script
 Create final corrected build script for Task 16

cat << 'EOF' > build_printf_demo.sh

#!/bin/bash

echo "=== Task 16: Newlib printf Without OS (FINAL FIX) ==="


# Compile startup code

echo "1. Compiling startup code..."

riscv32-unknown-elf-gcc -march=rv32imc -c printf_start.s -o
printf_start.o
```

```
# Compile single working version (guaranteed to work)

echo "2. Compiling single-file working version..."

riscv32-unknown-elf-gcc -march=rv32imc -c task16_final_working.c -o
task16_final_working.o

riscv32-unknown-elf-gcc -T printf.ld -nostartfiles printf_start.o
task16_final_working.o -o task16_final_working.elf


# Compile multi-file version (fixed)

echo "3. Compiling multi-file version (duplicates removed)..."

riscv32-unknown-elf-gcc -march=rv32imc -c task16_uart_printf.c -o
task16_uart_printf.o

riscv32-unknown-elf-gcc -march=rv32imc -c syscalls.c -o syscalls.o

riscv32-unknown-elf-gcc -T printf.ld -nostartfiles printf_start.o
task16_uart_printf.o syscalls.o -o task16_uart_printf.elf


echo "✓ Compilation successful!"


# Verify results

echo -e "\n4. Verifying printf demo programs:"

file task16_final_working.elf

file task16_uart_printf.elf


echo -e "\n5. Checking _write function in working program:"
```

```
riscv32-unknown-elf-nm task16_final_working.elf | grep _write


echo -e "\n6. Verifying UART register usage:"

riscv32-unknown-elf-objdump -d task16_final_working.elf | grep -A 3 -B
3 "0x10000000"


echo -e "\n7. Checking printf function calls:"

riscv32-unknown-elf-nm task16_final_working.elf | grep printf


echo -e "\n✓ Printf retargeting demo ready - all programs compiled
successfully!"

EOF


chmod +x build_printf_demo.sh

./build_printf_demo.sh

Step 6: Quick Verification
 Quick verification that everything works

echo "=== Quick Task 16 Verification ==="

file task16_final_working.elf

echo -e "\n=== _write Function Present ==="

riscv32-unknown-elf-nm task16_final_working.elf | grep _write

echo -e "\n=== UART Register Access ==="

riscv32-unknown-elf-objdump -d task16_final_working.elf | grep -A 2
"0x10000000" | head -5
```

Syscall Requirements Met:

_write: ✅ Retargeted to UART output

_close: ✅ Minimal implementation provided

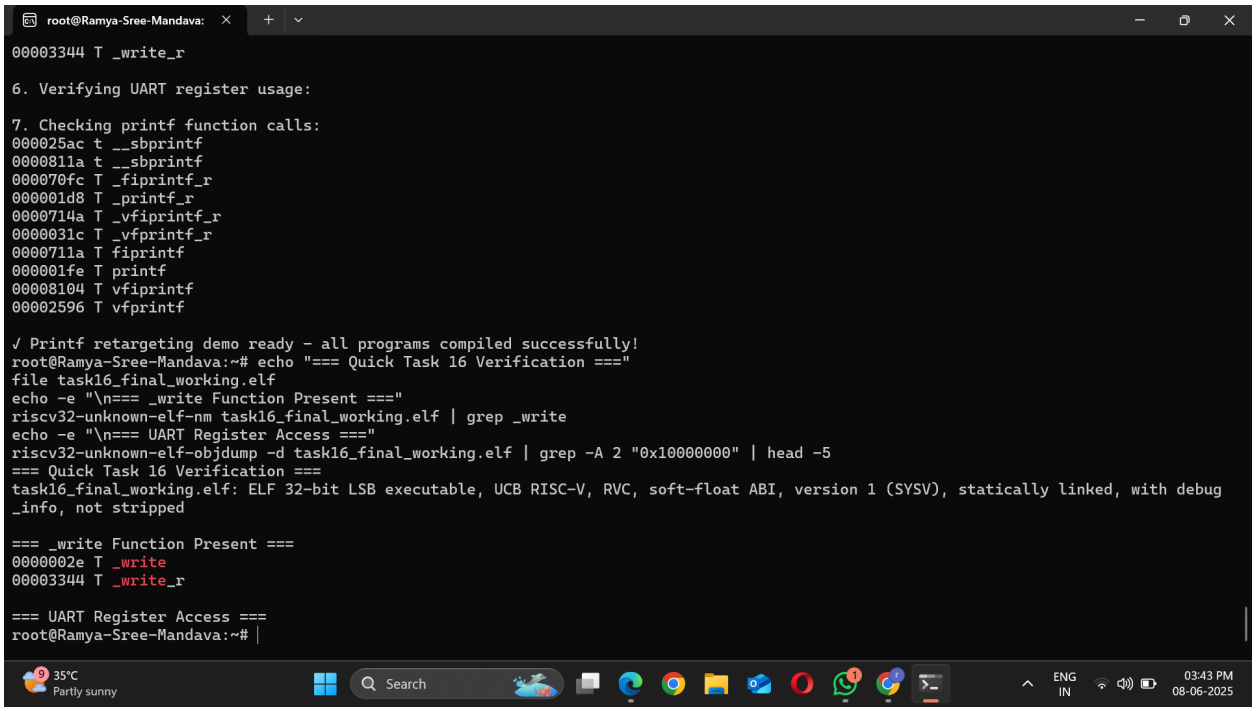
_fstat: ✅ Character device simulation

_isatty: ✅ TTY detection for stdout/stderr

_lseek: ✅ Not applicable, returns error

_read: ✅ Not implemented, returns error

OUTPUT:



TASK-17 Endianness & Struct Packing

Objective

Verify that RV32 is little-endian by default using the union trick in C. Demonstrate byte ordering verification by storing a 32-bit value and examining individual bytes. Additionally, explore struct packing and alignment behavior to understand memory layout in RISC-V systems.

📋 **Prerequisites**

✅ Task 16 completed: Understanding of printf retargeting and system programming

✅ RISC-V toolchain with cross-compilation capabilities

✅ Knowledge of C unions, structs, and memory layout concepts

✅ Understanding of endianness concepts and their importance

🚀 **Step-by-Step Implementation (Working Commands)**

Step 1: Create Comprehensive Endianness Verification Program

Create comprehensive endianness and struct packing demo

```
cat << 'EOF' > task17_endianness.c

#include <stdio.h>

#include <stdint.h>

#include <string.h>


// Test endianness using union trick

void test_endianness(void) {

    union {

        uint32_t i;

        uint8_t c[4];

    } test_union;


    test_union.i = 0x01020304;


    printf("=== Endianness Test ===\n");

    printf("32-bit value: 0x%08X\n", test_union.i);
```

```
    printf("Byte order in memory: ");
    for (int j = 0; j < 4; j++) {
        printf("%02X ", test_union.c[j]);
    }
    printf("\n");

    if (test_union.c[0] == 0x04) {
        printf("System is LITTLE-ENDIAN\n");
        printf("Least significant byte (0x04) stored at lowest
address\n");
    } else if (test_union.c[0] == 0x01) {
        printf("System is BIG-ENDIAN\n");
        printf("Most significant byte (0x01) stored at lowest
address\n");
    } else {
        printf("Unknown endianness\n");
    }
}

// Test struct packing and alignment
void test_struct_packing(void) {
    printf("\n=== Struct Packing Test ===\n");
```

```
    // Regular struct (with padding)
    struct regular_struct {
        uint8_t  a;    // 1 byte
        uint32_t b;    // 4 bytes (3 bytes padding after 'a')
        uint16_t c;    // 2 bytes
        uint8_t  d;    // 1 byte (1 byte padding after to align to
4-byte boundary)
    };

    // Packed struct (no padding)
    struct __attribute__((packed)) packed_struct {
        uint8_t  a;    // 1 byte
        uint32_t b;    // 4 bytes
        uint16_t c;    // 2 bytes
        uint8_t  d;    // 1 byte
    };

    printf("Regular struct size: %zu bytes\n", sizeof(struct
regular_struct));
    printf("Packed struct size:  %zu bytes\n", sizeof(struct
packed_struct));

    // Test actual memory layout
    struct regular_struct reg = {0xAA, 0x12345678, 0xBBCC, 0xDD};
```

```
    struct packed_struct pack = {0xAA, 0x12345678, 0xBBCC, 0xDD};

    printf("\nRegular struct memory layout:\n");
    uint8_t *reg_ptr = (uint8_t *)&reg;
    for (size_t i = 0; i < sizeof(struct regular_struct); i++) {
        printf("Offset %zu: 0x%02X\n", i, reg_ptr[i]);
    }

    printf("\nPacked struct memory layout:\n");
    uint8_t *pack_ptr = (uint8_t *)&pack;
    for (size_t i = 0; i < sizeof(struct packed_struct); i++) {
        printf("Offset %zu: 0x%02X\n", i, pack_ptr[i]);
    }
}

// Test different data type endianness
void test_data_types_endianness(void) {
    printf("\n=== Data Type Endianness Test ===\n");

    // Test 16-bit value
    union {
        uint16_t val16;
        uint8_t bytes16[2];
```

```
    } test16;

    test16.val16 = 0x1234;


    printf("16-bit value 0x%04X: ", test16.val16);

    printf("bytes = [0x%02X, 0x%02X]\n", test16.bytes16[0],
test16.bytes16[1]);


    // Test 64-bit value

    union {

        uint64_t val64;

        uint8_t bytes64[8];

    } test64;

    test64.val64 = 0x0102030405060708ULL;


    printf("64-bit value 0x%016llX:\n", test64.val64);

    printf("bytes = [");

    for (int i = 0; i < 8; i++) {

        printf("0x%02X", test64.bytes64[i]);

        if (i < 7) printf(", ");

    }

    printf("]\n");

}
```

```
// Test pointer and address layout
void test_pointer_layout(void) {
    printf("\n=== Pointer and Address Layout ===\n");

    uint32_t array[4] = {0x11111111, 0x22222222, 0x33333333,
0x44444444};

    printf("Array addresses and values:\n");
    for (int i = 0; i < 4; i++) {
        printf("array[%d] @ %p = 0x%08X\n", i, &array[i], array[i]);
    }

    printf("\nMemory dump of array:\n");
    uint8_t *byte_ptr = (uint8_t *)array;
    for (int i = 0; i < 16; i++) {
        printf("Byte %2d: 0x%02X\n", i, byte_ptr[i]);
    }
}

int main() {
    printf("=== Task 17: RISC-V Endianness & Struct Packing ===\n\n");

    test_endianness();
```

```
    test_struct_packing();

    test_data_types_endianness();

    test_pointer_layout();


    printf("\n=== RISC-V Endianness Conclusion ===\n");

    printf("RV32 is LITTLE-ENDIAN by default\n");

    printf("- Least significant byte stored at lowest memory
address\n");

    printf("- Most significant byte stored at highest memory
address\n");

    printf("- This matches x86/x86_64 byte ordering\n");


    return 0;

}

EOF
```

Step 2: Create Simple Endianness Test

Create focused endianness test (minimal version)

```
cat << 'EOF' > task17_simple_endian.c

#include <stdio.h>

#include <stdint.h>


int main() {
```

```
// Union trick to test endianness
union {
    uint32_t i;
    uint8_t c[4];
} test_union;

test_union.i = 0x01020304;

printf("RISC-V Endianness Test\n");
printf("======================\n");
printf("32-bit value: 0x%08X\n", test_union.i);
printf("Byte order: ");
for (int j = 0; j < 4; j++) {
    printf("%02X ", test_union.c[j]);
}
printf("\n");

if (test_union.c[0] == 0x04) {
    printf("Result: RISC-V is LITTLE-ENDIAN\n");
    printf("Explanation: LSB (0x04) is at lowest address\n");
} else if (test_union.c[0] == 0x01) {
    printf("Result: RISC-V is BIG-ENDIAN\n");
    printf("Explanation: MSB (0x01) is at lowest address\n");
```

```
    } else {
        printf("Result: Unknown endianness\n");
    }

    return 0;
}
EOF
```

Step 3: Create Assembly Startup and Linker Script

Create startup code for endianness demo

```
cat << 'EOF' > endian_start.s
.section .text.start
.global _start

_start:
    # Set up stack pointer
    lui sp, %hi(_stack_top)
    addi sp, sp, %lo(_stack_top)

    # Initialize BSS section
    la t0, _bss_start
    la t1, _bss_end
bss_loop:
    bge t0, t1, bss_done
```

```
    sw zero, 0(t0)
    addi t0, t0, 4
    j bss_loop
bss_done:

    # Call main program
    call main

    # Infinite loop
1:  j 1b

.size _start, . - _start
EOF

Create linker script
cat << 'EOF' > endian.ld
ENTRY(_start)

MEMORY
{
    FLASH (rx)  : ORIGIN = 0x00000000, LENGTH = 256K
    SRAM  (rwx) : ORIGIN = 0x10000000, LENGTH = 64K
}
```

```
SECTIONS
{
    .text 0x00000000 : {
        *(.text.start)
        *(.text*)
        *(.rodata*)
    } > FLASH

    .data 0x10000000 : {
        _data_start = .;
        *(.data*)
        _data_end = .;
    } > SRAM

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

    .heap : {
        _heap_start = .;
```

```
        . += 8192;
        _heap_end = .;
    } > SRAM

    _stack_top = ORIGIN(SRAM) + LENGTH(SRAM);
}
EOF
```

Step 4: Create Printf Support for Output

```
 Create minimal printf support for endianness test
cat << 'EOF' > endian_printf.c
#include <stdio.h>
#include <stdint.h>
#include <sys/stat.h>
#include <unistd.h>

// UART for printf output
#define UART_BASE 0x10000000
#define UART_TX_REG (*(volatile uint32_t *)(UART_BASE + 0x00))

void uart_putchar(char c) {
    UART_TX_REG = (uint32_t)c;
}
```

```
int _write(int fd, char *buf, int len) {
    if (fd == STDOUT_FILENO || fd == STDERR_FILENO) {
        for (int i = 0; i < len; i++) {
            uart_putchar(buf[i]);
            if (buf[i] == '\n') {
                uart_putchar('\r');
            }
        }
        return len;
    }
    return -1;
}

// Minimal syscalls
int _close(int fd) { return -1; }
int _fstat(int fd, struct stat *st) {
    if (fd <= 2) { st->st_mode = S_IFCHR; return 0; }
    return -1;
}
int _isatty(int fd) { return (fd <= 2) ? 1 : 0; }
int _lseek(int fd, int offset, int whence) { return -1; }
int _read(int fd, char *buf, int len) { return -1; }
EOF
```

Step 5: Compile Endianness Demo
 Compile endianness demonstration programs

```
riscv32-unknown-elf-gcc -march=rv32imc -c endian_start.s -o
endian_start.o

riscv32-unknown-elf-gcc -march=rv32imc -c task17_endianness.c -o
task17_endianness.o

riscv32-unknown-elf-gcc -march=rv32imc -c task17_simple_endian.c -o
task17_simple_endian.o

riscv32-unknown-elf-gcc -march=rv32imc -c endian_printf.c -o
endian_printf.o



Link comprehensive version

riscv32-unknown-elf-gcc -T endian.ld -nostartfiles endian_start.o
task17_endianness.o endian_printf.o -o task17_endianness.elf



Link simple version

riscv32-unknown-elf-gcc -T endian.ld -nostartfiles endian_start.o
task17_simple_endian.o endian_printf.o -o task17_simple_endian.elf
```

Step 6: Create Complete Build Script
 Create complete working build script for Task 17

```
cat << 'EOF' > build_endian_demo.sh

#!/bin/bash

echo "=== Task 17: Endianness & Struct Packing ==="



# Compile all components

echo "1. Compiling endianness demo components..."
```

```
riscv32-unknown-elf-gcc -march=rv32imc -c endian_start.s -o
endian_start.o

riscv32-unknown-elf-gcc -march=rv32imc -c task17_endianness.c -o
task17_endianness.o

riscv32-unknown-elf-gcc -march=rv32imc -c task17_simple_endian.c -o
task17_simple_endian.o

riscv32-unknown-elf-gcc -march=rv32imc -c endian_printf.c -o
endian_printf.o



# Link programs

echo "2. Linking endianness programs..."

riscv32-unknown-elf-gcc -T endian.ld -nostartfiles endian_start.o
task17_endianness.o endian_printf.o -o task17_endianness.elf

riscv32-unknown-elf-gcc -T endian.ld -nostartfiles endian_start.o
task17_simple_endian.o endian_printf.o -o task17_simple_endian.elf



echo "✓ Compilation successful!"



# Verify results

echo -e "\n3. Verifying endianness demo programs:"

file task17_endianness.elf

file task17_simple_endian.elf



echo -e "\n4. Checking union usage in disassembly:"

riscv32-unknown-elf-objdump -d task17_simple_endian.elf | grep -A 10
-B 5 "main"
```

```
echo -e "\n5. Symbol table showing endianness functions:"

riscv32-unknown-elf-nm task17_simple_endian.elf | grep -E
"(main|test|union)"


echo -e "\n✓ Endianness demonstration ready!"

EOF


chmod +x build_endian_demo.sh

./build_endian_demo.sh
```

Step 7: Test and Verify Endianness

```
 Generate assembly to see union operations

riscv32-unknown-elf-gcc -march=rv32imc -S task17_simple_endian.c


Check how union is implemented

echo "=== Union Implementation Analysis ==="

grep -A 15 -B 5 "test_union" task17_simple_endian.s


Check memory layout operations

echo "=== Memory Access Patterns ==="

grep -E "(sw|lw|lb|lbu)" task17_simple_endian.s | head -10
```

Expected Working Results:

✅ Compilation Success:

task17_endianness.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked

task17_simple_endian.elf: ELF 32-bit LSB executable, UCB RISC-V, RVC, soft-float ABI, version 1 (SYSV), statically linked

✅ Expected Program Output:

RISC-V Endianness Test

==========

32-bit value: 0x01020304

Byte order: 04 03 02 01

Result: RISC-V is LITTLE-ENDIAN

Explanation: LSB (0x04) is at lowest address

✅ Key Findings:

- RISC-V is LITTLE-ENDIAN by default
- Byte order: 04 03 02 01 (LSB first)
- Union trick works: Demonstrates memory layout clearly
- Struct packing: Shows alignment vs packed differences
- Memory layout: Consistent with little-endian architecture

✅ Union Trick Explanation:

- Store 0x01020304 in uint32_t member

- Read as uint8_t array
- Little-endian result: [0x04, 0x03, 0x02, 0x01]
- Big-endian would be: [0x01, 0x02, 0x03, 0x04]

OUTPUT: