ChatGPT

# RISC-V Bare-Metal Development Journey

Welcome to my personal repository documenting Week 1 of a full bare-metal RISC-V learning experience. Rather than simply follow instructions, I chose to go deeper — understanding not just what each step does, but how, why, and where it might go wrong.

This journey is based on the `p1w1.pdf` guide (Week 1 of a structured RISC-V course), but I've reimagined every task in a way that's hands-on, developer-friendly, and enriched with practical explanations, gotchas, and insights.

> ⚠ This is not a copy of the original guide. It's a ground-up reinterpretation with added commentary, real-world context, debugging notes, and extra learning layers.

---

## 🔍 What You'll Find

Each of the 17 tasks is carefully broken down into: - **Objectives** — What you're trying to achieve - 🛠 **Environment** — Tools or platforms involved - ✎ **Step-by-step instructions** — With clean code, commands, and order - **Expected outputs** — So you know you're on track - ⚠ **Common errors and fixes** — Because mistakes are part of learning - **What not to do** — Based on personal trial-and-error - **Pro tips** — Insights I wish I knew earlier - **Reflections** — What I actually learned at the end

You can think of this as the kind of walkthrough I would want to stumble upon when learning something new — not too formal, but not too shallow either.

---

### Repository Table of Contents

| Task No. | Title | Focus |
|---|---|---|
| **1** | Install Toolchain | Manual toolchain setup and environment validation |
| **2** | Compile Hello | Basic cross-compilation for RISC-V |
| **3** | C to Assembly | Understand how high-level code maps to instructions |
| **4** | Disassemble & Hex | View machine instructions and generate Intel HEX |
| **5** | RISC-V ABI | Learn register naming conventions and usage |
| **6** | Debug with GDB | Step-by-step debugging of RISC-V programs |
| **7** | Run on Emulator | Use Spike and QEMU to execute ELF files |
| **8** | Compiler Optimizations | Compare how `-O0`, `-O1`, `-O2`, `-O3` change output |

| Task No. | Title | Focus |
|---|---|---|
| **9** | Inline Assembly | Embed RISC-V instructions inside C code |
| **10** | CSRs | Read and write control/status registers |
| **11** | Memory-Mapped I/O | Use `volatile` pointers to control hardware |
| **12** | Linker Script | Define memory layout and section placement |
| **13** | Startup Code | Create `crt0.s` to set up stack and call main |
| **14** | Interrupts | Setup and test basic ISR with trap handler |
| **15** | Atomics & Fences | Synchronize memory and enforce order |
| **16** | `printf()` with Newlib | Enable printing without an OS |
| **17** | Endianness | Validate memory byte order using inspection |

---

## 📦 How to Use This Repo

```
git clone https://github.com/<your-username>/riscv-week1-tasks.git
cd riscv-week1-tasks
```

Each task lives in the `README.md` with all commands runnable directly on: - Ubuntu 22.04 LTS (or WSL2) - RISC-V GNU Toolchain (prebuilt) - Optional: Spike, QEMU, GDB, Makefiles

---

## 🛠 Tools & Resources Used

- `riscv32-unknown-elf-gcc`, `objdump`, `gdb`, `objcopy`, `readelf`
- QEMU: `qemu-system-riscv32`
- Spike emulator from `riscv-isa-sim`
- Visual Studio Code + `code .`
- Makefiles and `.ld` linker scripts

---

## Extra Features in This Guide

- Every task works offline — perfect for air-gapped development
- Commands are cut-and-paste ready
- Many steps include *expected* vs *actual* output comparisons
- 🪤 I've intentionally left a few "wrong ways" to help you see what failure looks like — and how to debug it

---

### Final Thoughts

This Week 1 journey helped me appreciate the low-level world of embedded systems, compilers, memory layout, and boot-time behavior. I no longer see `main()` as the beginning of a program — it’s just the part we usually notice.

> If you're serious about embedded, this kind of bare-metal work builds real confidence.

Feel free to fork, clone, and use this repo as your own personal training ground.

---

Week 2 will expand into linker memory regions, trap cause decoding, embedded `malloc`, interrupt vectors, and more advanced peripherals.

---

Let’s dive into the tasks! 👇

Welcome to my personal RISC-V assignment log. This guide is based on foundational learning tasks (Week 1) but redesigned in a style that's lighter, more reflective, and developer-oriented. The aim is to track what I did, why it worked (or didn’t), and what I learned.

---

## Task 1: Setting Up RISC-V Tools Without Pain

### Objective

To install the RISC-V toolchain and verify that all necessary compiler and debugger tools are accessible from the terminal.

### 🛠 Environment

- Ubuntu 22.04 LTS (WSL)
- Prebuilt RISC-V toolchain archive

### 🪜 Steps to Follow

#### Step 1: Unpack the Toolchain Archive

```
cd ~/Downloads
ls -lh riscv-toolchain-*.tar.gz
sudo mkdir -p /opt/riscv
sudo tar -xzf riscv-toolchain-*.tar.gz -C /opt/riscv --strip-components=1
```

This extracts the contents cleanly to `/opt/riscv`, keeping it isolated and organized.

Expected Output: - `/opt/riscv/bin` should now contain binaries like `riscv32-unknown-elf-gcc`

---

### Step 2: Add Toolchain to PATH

```
echo 'export PATH=/opt/riscv/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
which riscv32-unknown-elf-gcc
```

This allows you to call RISC-V tools from any directory.

Expected Output: - `/opt/riscv/bin/riscv32-unknown-elf-gcc`

---

### Step 3: Confirm Toolchain Installation

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

You should see version information if the setup is correct.

Expected Output: - Version banners from GCC, GDB, and Objdump

---

## ⚠ Common Pitfalls

| Issue | Cause | Fix |
|---|---|---|
| `command not found` | PATH not sourced | Run `source ~/.bashrc` |
| `Permission denied` | Missing `sudo` | Add `sudo` before `tar` |
| Wrong GCC invoked | Conflicting path order | Use `which` to verify location |

### Avoid This

- Don't extract directly in `Downloads`
- Don't forget to permanently update your PATH

### Pro Tips

- Use `command -v` to confirm binary presence
- Backup `.bashrc` before changes

### Reflection

> Learned how toolchains are set up and validated, and how `$PATH` controls command availability.

---

## ✍ Task 2: Compile Your First RISC-V Program

### Objective

Cross-compile a basic C program to verify the RISC-V GCC is working.

### 🛠 Environment

- RISC-V GCC
- `file` command for ELF type checking

### 🪜 Steps to Follow

#### Step 1: Create `hello.c`

```c
#include <stdio.h>
int main() {
    printf("Hello, RISC-V!\n");
    return 0;
}
```

Save this file inside your working folder.

> This is the source you'll compile using the cross-compiler.

---

#### Step 2: Compile to ELF

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

> If this succeeds without error, the compiler is working.

Expected Output: - File `hello.elf` created

---

#### Step 3: Inspect the Binary

```
file hello.elf
```

Confirms architecture target

Expected Output:

```
hello.elf: ELF 32-bit LSB executable, UCB RISC-V, version 1 (SYSV)
```

---

## ⚠ Common Pitfalls

| Issue | Cause | Fix |
|---|---|---|
| `cannot execute binary file` | Trying to run ELF on x86 | Use QEMU or spike |
| Unknown arch error | Missing flags | Always include `-march` and `-mabi` |

### Avoid This

- Don't try to run the `.elf` directly on your PC
- Don't omit architecture/ABI flags on embedded targets

### Pro Tips

- Run `readelf -h hello.elf` to learn more about the ELF header
- Use `-Wall -Werror` to make GCC strict

### Reflection

> It's satisfying to compile for a completely different architecture from x86. Learned the power of cross-compilers.

---

# Task 3: From C to Assembly — Understanding the Generated Code

### Objective

Convert the compiled C program into RISC-V assembly code and break down the function prologue and epilogue.

## 🛠 Environment

- `riscv32-unknown-elf-gcc`
- Source file: `hello.c`

## 🪜 Steps to Follow

### Step 1: Generate Assembly from C

Use `-S` to stop at the assembly stage, and `-O0` to disable optimizations.

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

This creates `hello.s` — the human-readable RISC-V assembly version of your C code.

Expected Output: - A new file named `hello.s` containing the generated RISC-V assembly

---

### Step 2: Inspect the Assembly File

Open the `.s` file and find the main function prologue:

```
addi    sp,sp,-16
sw      ra,12(sp)
sw      s0,8(sp)
addi    s0,sp,16
```

These instructions set up the stack frame and save important registers.

Breakdown: - `addi sp,sp,-16` → allocate stack space - `sw ra,12(sp)` → store return address - `sw s0,8(sp)` → store old frame pointer - `addi s0,sp,16` → set new frame pointer

---

## ⚠ Common Pitfalls

| Issue | Cause | Fix |
|---|---|---|
| Empty or missing `.s` file | Wrong input or path | Double-check filenames and rerun with `-S` |
| Confusing output | Optimization enabled | Use `-O0` to disable transformation |

### Avoid This

- Don't use high optimization levels when trying to study raw structure

- Don’t overwrite your source accidentally (e.g., by naming `.s` as `.c`)

### Pro Tips

- Use `-fverbose-asm` to include helpful comments in the assembly
- Try different `-O` levels (`-O1`, `-O2`) to compare output

### Reflection

> Looking at the assembly shows what happens behind a simple `main()`. It's cool to see stack setup and register saves up close.

---

## Task 4: Disassembly and Hex Dump — Viewing Machine Instructions

### Objective

To disassemble your compiled ELF file into readable instructions and generate a hex dump suitable for simulators or flashing.

### 🛠 Environment

- `riscv32-unknown-elf-objdump`
- `riscv32-unknown-elf-objcopy`
- Input ELF: `hello.elf`

### 🪜 Steps to Follow

#### Step 1: Disassemble the ELF File

Use `objdump` to get the full instruction view:

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

This creates a `.dump` file with assembly instructions and addresses.

Expected Output: - `hello.dump` contains lines like:

```
00000000 <main>:
   0:   1141        addi    sp,sp,-16
   2:   c606        sw      ra,12(sp)
```

---

### Step 2: Convert ELF to Intel HEX Format

Use `objcopy` to generate a raw hex file:

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

This creates a text-based memory image, useful for emulators.

Expected Output: - `hello.hex` is a readable hex file in Intel HEX format - Lines look like: `:10C000000000002001000000...`

---

## ⚠ Common Pitfalls

| Issue | Cause | Fix |
|---|---|---|
| `no such file` | Misspelled filename | Check if `hello.elf` exists |
| Empty hex file | Incorrect ELF format | Recompile using `-march=rv32imc` |

### Avoid This

- Don’t use `.hex` with missing sections — check `objdump` first
- Don’t forget `-O ihex` or the output will be unusable

### Pro Tips

- Use `less hello.dump` to scroll through large disassembly
- Try `-S` flag with `objdump` to interleave source and assembly

### Reflection

> Seeing real binary instructions helped bridge the gap between high-level C and machine-level encoding. It’s like peeling back software’s outer layers.

---

# Task 5: Understanding the RISC-V ABI and Registers

### Objective

Familiarize yourself with the RISC-V integer register set, ABI naming, and calling convention roles.

## 🛠 Environment

- No tools required — purely conceptual understanding
- Optional command: `riscv32-unknown-elf-gdb` to inspect registers live

## Register Overview

RISC-V has 32 general-purpose integer registers:

| Register | ABI Name | Description |
|---|---|---|
| x0 | zero | Constant 0 |
| x1 | ra | Return address |
| x2 | sp | Stack pointer |
| x3 | gp | Global pointer |
| x4 | tp | Thread pointer |
| x5–x7 | t0–t2 | Temporaries (caller-saved) |
| x8 | s0/fp | Saved/frame pointer (callee-saved) |
| x9 | s1 | Saved register |
| x10–x17 | a0–a7 | Argument registers / return values |
| x18–x27 | s2–s11 | Saved registers (callee-saved) |
| x28–x31 | t3–t6 | More temporaries |

---

### Calling Convention Rules

- **Arguments** are passed using `a0` to `a7`
- **Return values** come back in `a0`, `a1`
- **Caller-saved**: Temporaries (`t0–t6`) — must be preserved by caller if needed
- **Callee-saved**: Saved registers (`s0–s11`) — must be preserved by callee

---

## Optional Exploration

If you want to see these in action using `gdb`:

```
gdb hello.elf
(gdb) target sim
(gdb) break main
(gdb) run
(gdb) info registers
```

This shows values in each register at runtime.

Expected Output: - Register table with values assigned (e.g., `a0`, `sp`, `ra` etc.)

---

### ⚠ Common Pitfalls

| Misunderstanding | Clarification |
|---|---|
| Confusing `s` and `t` roles | `s` registers are preserved by functions; `t` are not |
| Thinking `x0` is writable | It’s hardwired to zero and always reads as 0 |

#### Avoid This

- Don’t assume RISC-V register naming follows x86 patterns
- Don’t write to `x0` — it's always zero

#### Pro Tips

- Memorize argument and return register roles (`a0–a7`)
- Use `info reg` in `gdb` to watch register changes dynamically

#### Reflection

> Understanding the register map helped demystify how data flows during function calls. It clarified what gets preserved and who’s responsible for what.

---

## Task 6: Debugging with GDB — Breakpoints and Stepping Through Code

#### Objective

Use the RISC-V version of GDB to load the ELF file, set breakpoints, step through instructions, and examine register values in simulation.

### 🛠 Environment

- `riscv32-unknown-elf-gdb`
- Compiled ELF file: `hello.elf`

### 🪜 Steps to Follow

#### Step 1: Start GDB with Your ELF

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

This loads the binary into GDB so you can debug it interactively.

Expected Output: - GDB prompt `gdb>` appears

---

### Step 2: Start the Simulator Backend

```
target sim
```

This runs the program in a simulated environment.

Expected Output: - Message confirming target simulation connected

---

### Step 3: Set a Breakpoint at `main`

```
break main
```

This sets a breakpoint so the program halts at the beginning of `main()`.

Expected Output: - `Breakpoint 1 at ...` message shown

---

### Step 4: Run the Program

```
run
```

The execution starts and then halts at the breakpoint in `main()`.

Expected Output: - Output confirming breakpoint hit

---

### Step 5: Step Through Instructions

```
stepi   # step a single instruction
info registers
```

You can now trace exactly what each instruction does and view register values.

Expected Output: - Current instruction, and full register dump

---

## ⚠ Common Pitfalls

| Issue | Cause | Fix |
|---|---|---|
| `target sim` fails | Wrong GDB or ELF | Use ELF compiled with bare-metal GCC |
| Breakpoint not hit | Function optimized away | Compile with `-00` to preserve symbols |

### Avoid This

- Don’t use the wrong GDB binary (must be RISC-V GDB)
- Don’t skip setting the target or nothing will run

### Pro Tips

- Use `layout asm` for a split view if in a terminal with ncurses
- Use `disassemble` to see the instructions in memory

### Reflection

> This step taught me how to go instruction-by-instruction in a simulated RISC-V environment. GDB is powerful for seeing exactly what’s happening under the hood.

---

# 🖥 Task 7: Running RISC-V Programs on Spike and QEMU

### Objective

Run your compiled ELF on a RISC-V emulator to simulate bare-metal execution and verify output (such as UART print behavior).

## 🛠 Environment

- Spike (`spike`)
- QEMU (`qemu-system-riscv32`)
- ELF binary: `hello.elf`

## 🪜 Steps to Follow

#### Step 1: Run Using Spike (if installed)

```
spike --isa=rv32imc pk hello.elf
```

Spike launches a proxy kernel and executes your ELF.

Expected Output: - Console prints from the program (if `printf` routes to proxy kernel)

Note: You must have `pk` (proxy kernel) from riscv-pk installed and in your PATH.

---

**Step 2: Run Using QEMU (no proxy kernel needed)**

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

QEMU runs a headless RISC-V system and executes your program.

Expected Output: - Any `printf` output appears directly in the terminal (UART-style)

---

## ⚠ Common Pitfalls

| Issue | Cause | Fix |
|---|---|---|
| `spike: command not found` | Not installed | Build from riscv-tools or install via package manager |
| `qemu: unsupported machine` | Wrong or missing machine config | Use `-machine sifive_e` or similar |
| No output | Missing semihosting or UART support | Ensure `printf` maps to standard I/O or UART |

### Avoid This

- Don’t confuse `qemu-system-x86` with `qemu-system-riscv32`
- Don’t forget `-nographic` in QEMU — or you won't see output

### Pro Tips

- Use `-S -s` flags in QEMU to connect a debugger
- You can install QEMU via `sudo apt install qemu-system-misc`

### Reflection

> Running RISC-V code without hardware makes development practical. Seeing `printf` output appear via UART on QEMU was especially rewarding.

---

# ⚙ Task 8: Exploring Compiler Optimizations in RISC-V GCC

## Objective

Observe how different optimization levels affect the generated assembly code, binary size, and performance.

## 🛠 Environment

- `riscv32-unknown-elf-gcc`
- Input: `hello.c`

## 🪜 Steps to Follow

### Step 1: Compile with No Optimization

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

Produces verbose, unoptimized assembly — helpful for learning.

Expected Output: - `hello_O0.s` contains raw instruction mappings without transformations

---

### Step 2: Compile with Higher Optimizations

```
riscv32-unknown-elf-gcc -O1 -S hello.c -o hello_O1.s
riscv32-unknown-elf-gcc -O2 -S hello.c -o hello_O2.s
riscv32-unknown-elf-gcc -O3 -S hello.c -o hello_O3.s
```

Generates optimized assembly for each level of aggressiveness.

Expected Output: - `.s` files get progressively smaller, tighter, and potentially harder to read

---

### Step 3: Compare the Outputs

Use `diff` or `code` editors to visually inspect changes:

```
diff hello_O0.s hello_O2.s
```

You'll see reduced instruction count, loop unrolling, and removed redundancies at higher `-O` levels.

---

### ⚠ Common Pitfalls

| Problem | Cause | Solution |
|---|---|---|
| No changes seen | Compiler cache or minor code | Use complex logic to see more optimizations |
| Missing labels | Optimizer may inline/remove functions | Use `-fno-inline` if needed for debugging |

### Avoid This

- Don't assume higher optimization is always better — `-O3` can increase code size
- Don't analyze behavior at `-O2` expecting `-O0` style structure

### Pro Tips

- Add `-fverbose-asm` to include compiler comments in output
- Use `size hello.elf` to compare code/data section sizes across builds

### Reflection

> Optimization levels reveal how compilers enhance performance by rewriting and reducing code. It helped me appreciate why high-level code isn't always what runs on hardware.

---

## Task 9: Inline Assembly — Embedding RISC-V Instructions in C

### Objective

Learn to embed basic RISC-V instructions into C code using the `asm` keyword. Understand how to move values between registers and apply simple arithmetic using inline assembly.

### 🛠 Environment

- RISC-V GCC with inline assembly support
- Source file: `inline_asm.c`

### 📏 Steps to Follow

#### Step 1: Write C Code with Inline Assembly

```
int result;
int main() {
    int a = 10, b = 20;
    asm volatile ("add %0, %1, %2" : "=r"(result) : "r"(a), "r"(b));
    return 0;
}
```

This program uses inline assembly to compute `result = a + b`.

Expected Output: - Successful compilation - If inspected with `objdump`, the `add` instruction appears in the output

---

**Step 2: Compile the Program**

```
riscv32-unknown-elf-gcc -S -O0 inline_asm.c -o inline_asm.s
```

Generates the `.s` file so we can verify if the `add` instruction is properly embedded.

Expected Output: - `inline_asm.s` should contain: `add a5, a4, a3` (or similar)

---

## ⚠ Common Pitfalls

| Problem | Cause | Fix |
|---|---|---|
| Registers don't appear as expected | Compiler optimization may change them | Use `-O0` and `volatile` to preserve structure |
| Invalid operand constraints | Misuse of `"r"`, `"=r"` | Double-check constraint syntax |

### Avoid This

- Don't forget `volatile` if you want to prevent compiler optimization
- Don't try to do multi-line or branching logic inside inline asm without understanding constraints

### Pro Tips

- Use `"r"` to let the compiler pick any general-purpose register
- Try adding comments in the `asm` block for readability

### Reflection

> Writing inline assembly deepens your understanding of register-level operations and how compilers pass values to instructions. It's a powerful tool for performance-critical code.

---

# Task 10: Accessing RISC-V CSRs — Control and Status Registers

## Objective

Learn how to read and write RISC-V Control and Status Registers (CSRs) using inline assembly. Understand how CSRs manage core hardware features like counters, machine mode, interrupts, etc.

## 🛠 Environment

- RISC-V GCC
- Inline assembly in C (`asm volatile`) and prebuilt CSR names

## 🪜 Steps to Follow

### Step 1: Create a CSR Read/Write Program

```
unsigned int mcycle_val;
int main() {
    asm volatile ("csrr %0, mcycle" : "=r"(mcycle_val));
    return 0;
}
```

This example reads the current cycle count from the `mcycle` CSR and stores it in `mcycle_val`.

Expected Output: - Valid compilation - Inspecting the `.s` file should show a `csrr` instruction

---

### Step 2: Compile and View Assembly

```
riscv32-unknown-elf-gcc -O0 -S csr_read.c -o csr_read.s
less csr_read.s
```

Look for this line:

```
csrr a5, mcycle
```

---

### Step 3: Modify a CSR (if applicable)

```
int main() {
    unsigned int dummy = 0;
    asm volatile ("csrw 0x7C0, %0" :: "r"(dummy)); // writes to custom CSR
    return 0;
}
```

This is a raw way to write to a CSR using its hex address (e.g., 0x7C0).

---

## ⚠ Common Pitfalls

| Problem | Cause | Fix |
|---|---|---|
| `unknown CSR` | Misspelled name or unsupported CSR | Use official RISC-V CSR list or try numeric CSR ID |
| `Illegal instruction` on run | CSR not supported in simulator | Use spike or a real RISC-V core with CSR support |

### Avoid This

- Don’t assume all CSRs are accessible in user-mode programs
- Don’t write to CSRs that could disrupt machine state (e.g., `mstatus`, `mepc`) without caution

### Pro Tips

- Use symbolic names like `mcycle` or `mhartid` for readability
- Use `objdump -d` to verify generated `csrr/csrw` opcodes

### Reflection

> This task gave me visibility into how processors manage internal state and performance counters. It's low-level but important for any bare-metal work.

---

# Task 11: Memory-Mapped I/O Using Volatile Pointers

### Objective

To access hardware registers via memory-mapped I/O using volatile pointers in C. This is foundational for interacting with real or simulated peripherals in embedded systems.

## 🛠 Environment

- Bare-metal C code
- RISC-V GCC

## 🪜 Steps to Follow

#### Step 1: Define a Volatile Pointer to a Memory Address

```
#define GPIO_CTRL_ADDR 0x10000000

int main() {
    volatile unsigned int *gpio_ctrl = (unsigned int *) GPIO_CTRL_ADDR;
    *gpio_ctrl = 0x1; // write to GPIO control register
    return 0;
}
```

This line tells the compiler: "don’t optimize away this access — it might do something meaningful."

Expected Output: - Successful compilation - `objdump` shows a `sw` (store word) to address `0x10000000`

---

**Step 2: Compile and Verify Memory Access**

```
riscv32-unknown-elf-gcc -O0 -S mmio.c -o mmio.s
```

This generates assembly code — useful for confirming correct `sw` instruction

Expected Assembly Snippet:

```
sw a5, 0(a4)
```

- With `a4` containing `0x10000000`

---

## ⚠ Common Pitfalls

| Problem | Cause | Fix |
|---|---|---|
| Nothing happens | You're not on a platform where that address is mapped | Test with simulator or on real hardware |
| Optimized away | Forgot `volatile` | Always use `volatile` for memory-mapped I/O |

### Avoid This

- Don’t use normal pointers for I/O registers — it may be optimized out
- Don’t assume the address is valid without a datasheet or emulator config

### Pro Tips

- In real boards, GPIO addresses are documented in reference manuals

- Use `volatile` for any register access that might change outside the CPU’s view

### Reflection

> This introduced me to how software reaches out to hardware. By just writing to an address, we control real-world devices — a powerful embedded principle.

---

## Task 12: Linker Script Basics and Memory Layout

### Objective

Understand how linker scripts define memory layout in embedded systems. Learn to create a custom `.ld` linker script and observe how it places code and data into specific memory addresses.

### 🛠 Environment

- RISC-V GCC
- Custom linker script (e.g., `link.ld`)
- Source file: `main.c`

### 🪜 Steps to Follow

#### Step 1: Create a Minimal Linker Script

```
ENTRY(_start)
SECTIONS {
  . = 0x80000000;
  .text : { *(.text*) }
  .rodata : { *(.rodata*) }
  .data : { *(.data*) }
  .bss : { *(.bss*) }
  /DISCARD/ : { *(.comment*) *(.note*) }
}
```

This script starts your program at address `0x80000000` and groups sections manually.

Expected Output: - You have a file `link.ld` ready to use with GCC

---

#### Step 2: Compile Using Your Custom Script

```
riscv32-unknown-elf-gcc -T link.ld -nostartfiles -o main.elf main.c
```

The `-T` flag tells GCC to use `link.ld`. The `-nostartfiles` avoids default startup.

Expected Output: - `main.elf` is created with memory layout matching `link.ld`

---

**Step 3: Inspect the ELF Layout**

```
riscv32-unknown-elf-readelf -S main.elf
```

This lists the sections, addresses, and sizes in your ELF file.

Expected Output:

```
[Nr] Name       Type     Addr     Off  Size
  .text        PROGBITS 80000000 ...
  .data        PROGBITS 8000xxxx ...
```

---

## ⚠ Common Pitfalls

| Issue | Cause | Fix |
|---|---|---|
| Sections missing | Incorrect linker syntax | Double-check `SECTIONS` block and entry point |
| Starts at 0x0 | No `. =` directive | Explicitly define start address |

### Avoid This

- Don’t rely on default scripts when learning layout — create your own
- Don’t forget `ENTRY(_start)` if you have no startup code

### Pro Tips

- Use `objdump -h main.elf` to see section offsets and addresses
- Separate memory (ROM, RAM) using MEMORY blocks if targeting real chips

### Reflection

> Linker scripts gave me insight into the memory structure of embedded binaries. It’s like blueprinting your code layout before deployment.

---

# Task 13: Writing Custom Startup Code (crt0.s)

## Objective

Manually write a minimal startup file (`crt0.s`) to initialize the stack and jump to `main`. This replaces the default runtime setup and gives full control over program boot.

## 🛠 Environment

- Assembly file: `crt0.s`
- Custom linker script from Task 12
- Bare-metal `main.c`

## 🪜 Steps to Follow

### Step 1: Write `crt0.s` for Startup

```
.section .text
.global _start
_start:
  la sp, _stack_top     # initialize stack pointer
  call main             # jump to C main
  j .                   # loop forever after main
```

This startup routine initializes the stack and transfers control to `main()`.

Expected Output: - The `_start` label will be the ELF entry point

---

### Step 2: Add Stack Symbol in Linker Script

In `link.ld`, define the stack top:

```
.stack (NOLOAD) : {
  . = ALIGN(4);
  _stack_top = .;
  . += 0x1000; /* 4KB stack */
} > RAM
```

Ensures `la sp, _stack_top` has a valid memory region

---

### Step 3: Compile Everything Together

```
riscv32-unknown-elf-gcc -T link.ld -nostdlib -nostartfiles crt0.s main.c -o
firmware.elf
```

`-nostartfiles` prevents default startup. `-nostdlib` disables linking to libc.

Expected Output: - `firmware.elf` uses your custom `_start` symbol

---

**Step 4: Disassemble and Inspect**

```
riscv32-unknown-elf-objdump -d firmware.elf
```

Confirm `_start` shows your custom instructions

Expected Output: - `_start:` block with `la sp, _stack_top` and `call main`

---

## ⚠ Common Pitfalls

| Problem | Cause | Fix |
|---|---|---|
| Stack not set | Missing or misaligned `_stack_top` | Add correct `.stack` section in linker |
| Jump to main fails | `main` not marked as global | Ensure `main` is correctly compiled and linked |

### Avoid This

- Don't forget to align stack properly (use `. = ALIGN(4)`)
- Don't call `main` without initializing `sp`

### Pro Tips

- Add `.globl main` if you write `main` in assembly
- Use a loop (`j .`) at end of `_start` to prevent falling off

### Reflection

> Writing `crt0.s` helped me understand what actually runs before `main()`. It's bare-metal to the core, and I now see how embedded programs boot up.

---

# ⚡ Task 14: Writing and Triggering a Basic Interrupt Service Routine (ISR)

## Objective

Learn how to create a minimal interrupt service routine (ISR) and configure your system to trigger it. Understand how exceptions and traps are handled in RISC-V.

## 🛠 Environment

- RISC-V toolchain
- Custom `crt0.s` and linker script
- Inline assembly and `mtvec` configuration

## 🪜 Steps to Follow

### Step 1: Define an ISR Function

```
void isr() {
    volatile int *led = (int *)0x10000000;
    *led = 0xFF;  // Example side effect
}
```

This will be the trap handler, showing side effects like setting a memory-mapped register.

---

### Step 2: Set the Trap Vector (in `main()` or assembly)

```
int main() {
    asm volatile("csrw mtvec, %0" :: "r"(isr));
    while (1);
}
```

Sets `mtvec` to point to your `isr()` handler.

---

### Step 3: Trigger the Trap

You can use software to trigger a trap:

```
asm volatile("ecall"); // Causes environment call exception
```

Triggers a trap, and control should jump to `isr()` if configured correctly.

Expected Output: - If `isr()` writes to memory, the value should update as expected (e.g., LED turns on)

---

## ⚠ Common Pitfalls

| Problem | Cause | Fix |
|---|---|---|
| ISR not called | `mtvec` not set properly | Use `csrw mtvec, handler_addr` in `main()` |
| Reset/reboot | Invalid memory access | Ensure ISR points to valid flash/ROM/RAM |

### Avoid This

- Don't forget to align the `mtvec` handler address (4-byte aligned)
- Don't call functions from within `isr()` unless stack is guaranteed

### Pro Tips

- Use `-nostartfiles` and `-nostdlib` to ensure your environment is bare-metal
- Label your `isr` with `__attribute__((interrupt))` for clarity and alignment

### Reflection

> Traps and ISRs are core to embedded responsiveness. Hooking one manually showed how control flow can jump to custom logic based on low-level triggers.

---

# Task 15: Atomic Operations and Memory Fences in RISC-V

### Objective

Understand how atomic instructions and memory fences work in RISC-V, and why they are critical for multi-core synchronization and memory visibility.

## 🛠 Environment

- RISC-V GCC
- Optional: multi-core simulator like Spike with multiple harts

## 🪜 Steps to Follow

#### Step 1: Write Code Using Atomic Instruction (`amoswap.w`)

```
volatile int lock = 0;
int main() {
    int my_id = 1;
    asm volatile("amoswap.w %0, %2, (%1)"
                 : "=r"(my_id)
                 : "r"(&lock), "r"(my_id)
                 : "memory");
    return 0;
}
```

This code atomically swaps `my_id` into the memory location `lock`, storing the old value in `my_id`.

Expected Output: - If `lock` was 0, `my_id` gets 0, and `lock` becomes 1

---

**Step 2: Add Memory Fences**

```
asm volatile("fence rw, rw" ::: "memory");
```

Ensures all reads/writes before the fence complete before any that follow it

Expected Output: - Behavior is correct and predictable even in multi-threaded settings

---

## ⚠ Common Pitfalls

| Problem | Cause | Fix |
|---|---|---|
| `Illegal instruction` | Not supported in ISA or emulator | Ensure you compile with `-march=rv32a` |
| Unpredictable results | No memory barrier | Use `fence` instructions for ordering |

### Avoid This

- Don’t test atomic instructions in single-core-only simulators without A-extension
- Don’t omit the `"memory"` clobber in inline assembly

### Pro Tips

- Use `fence` to prevent compiler/hardware reordering
- Use atomic ops to implement spinlocks or mutexes

### Reflection

This task helped me realize how delicate shared memory is. Without atomic ops and fences, data races are inevitable.

## 📤 Task 16: Using `printf()` with Newlib (No OS)

### Objective

Use `printf()` in a bare-metal RISC-V program without an operating system, by linking against Newlib and providing your own `write()` syscall stub.

### 🛠 Environment

- Newlib installed via toolchain
- RISC-V GCC
- Custom linker + `crt0.s`

### 🪜 Steps to Follow

#### Step 1: Write a Syscall Stub for `write()`

Newlib calls `write()` for `printf()`. You must define this:

```c
#include <unistd.h>
int write(int fd, const void *buf, size_t count) {
    const char *c = buf;
    for (size_t i = 0; i < count; i++) {
        volatile char *uart = (char *) 0x10000000;
        *uart = c[i];
    }
    return count;
}
```

This maps each character to a memory-mapped UART at `0x10000000`

Expected Output: - Printed strings via UART in emulator (e.g., QEMU)

---

#### Step 2: Use `printf()` in Main

```c
#include <stdio.h>
int main() {
    printf("Hello from bare-metal!
");
    return 0;
}
```

Now `printf()` is backed by your `write()` stub

---

**Step 3: Compile With Newlib**

```
riscv32-unknown-elf-gcc -T link.ld -nostartfiles -o printf.elf crt0.s main.c
write.c -lc -lgcc
```

Link manually against libc and libgcc, but omit standard startup files

Expected Output: - `printf.elf` ready for emulator — prints to UART if mapped

---

## ⚠ Common Pitfalls

| Problem | Cause | Fix |
|---|---|---|
| No output | `write()` not defined or UART not simulated | Define `write()` stub and test in QEMU |
| `printf` compile error | Newlib not linked | Add `-lc` and `-lgcc` manually |

### Avoid This

- Don’t rely on OS syscalls — they're not available in bare-metal
- Don’t forget to implement all weak syscalls (`write`, `exit`, etc.) if needed

### Pro Tips

- Add `-Wl,-Map=out.map` to see how symbols are resolved
- In QEMU, use `-nographic -serial mon:stdio` to see UART output

### Reflection

Making `printf()` work with just a `write()` stub showed how minimal a working C runtime can be. You don't need an OS — just a UART and some glue code.

---

# Task 17: Endianness Testing and Validation

## Objective

Verify whether your RISC-V system uses little-endian or big-endian format by writing and reading known memory patterns.

## 🛠 Environment

- RISC-V GCC
- Custom C code
- Tools: `objdump`, `gdb`, or memory inspection

## 🪜 Steps to Follow

### Step 1: Write a Known Pattern to Memory

```c
int main() {
    int val = 0x11223344;
    volatile char *ptr = (char *)&val;
    for (int i = 0; i < 4; i++) {
        // Inspect byte order
    }
    return 0;
}
```

This stores a 4-byte value, then lets us check how bytes appear in memory.

---

### Step 2: Dump Memory with GDB

```
riscv32-unknown-elf-gdb test.elf
(gdb) target sim
(gdb) break main
(gdb) run
(gdb) x/4xb &val
```

Use GDB to examine the 4 bytes at `val`'s address

Expected Output: - If little-endian: `0x44 0x33 0x22 0x11` - If big-endian: `0x11 0x22 0x33 0x44`

---

## ⚠ Common Pitfalls

| Problem | Cause | Fix |
|---|---|---|
| All bytes zero | Optimized away | Use `volatile` or compile with `-O0` |
| Memory misalignment | Wrong pointer cast | Always align properly, or use `uint8_t*` |

### Avoid This

- Don’t assume default endian — verify it for your toolchain and hardware
- Don’t forget to declare test variables as `volatile` to avoid optimization

### Pro Tips

- Use `hexdump` or `objdump -s` for binary-level inspection
- Most RISC-V systems default to **little-endian**, but check anyway

### Reflection

> This task made me think about how data is stored at the byte level. Knowing your system's endian behavior is essential when working with memory-mapped hardware, binary protocols, or cross-platform data exchange.

---

Task 17 complete. All Week 1 tasks are now redesigned, written, and documented!