EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 1a: Understanding the ARM Stack: Inline Assembly and Live Debugging

ARM Cortex-M33 ● GDB ● Inline Assembly

September 6, 2026

For educational and defensive security research only

LEGAL DISCLAIMER: The information, tools, and code provided in this repository and course are strictly for educational, research, and defensive purposes only.

You are explicitly prohibited from using any materials contained herein to access, test, modify, or exploit any device, network, or system that you do not own 100% or for which you do not have explicit, documented, and legally binding authorization to interact with.

By using this repository and course, you acknowledge and agree that:

1. Any illegal, unauthorized, or malicious use of this information is solely your responsibility.
2. The author(s) and contributor(s) of this repository and course shall not be held liable for any damages, legal repercussions, criminal charges, or unauthorized actions resulting from the use, misuse, or abuse of the contents herein.
3. You will comply with all applicable local, state, national, and international laws regarding cybersecurity and computer fraud.

IF YOU DO NOT AGREE WITH THESE TERMS, DO NOT USE THIS REPOSITORY AND COURSE.

What You'll Learn This Week

By the end of this week, you will be able to:

- Understand how the RP2350 Cortex-M33 stack grows in SRAM.
- Identify the ARM registers used by this stack experiment.
- Build and flash a Pico 2 ELF through a Debug Probe.
- Connect OpenOCD and GDB to live hardware.
- Step one assembly instruction at a time with `si`.
- Examine the exact stack words written by each multi-register instruction.
- Prove that an ARM register list is ordered by register number, not source-list spelling.

Part 1: Understanding the Basics

What is a Microcontroller?

A microcontroller is a complete small computer on one chip. It contains processor cores, memory controllers, peripherals, and interfaces for hardware such as GPIO, UART, timers, and SPI. The Raspberry Pi Pico 2 uses the **RP2350** microcontroller.

What is the ARM Cortex-M33?

The RP2350 can run Arm Cortex-M33 cores. The program in this folder is built for that Arm target. We will use the Debug Probe, OpenOCD, and GDB to stop a core and inspect its registers and memory while it executes the program.

What is Dynamic Analysis?

Dynamic analysis means observing a program while it runs on real hardware. In this lesson we will:

- Stop the processor at `main`.
- View the instructions produced by the compiler.
- Execute one instruction with `si`.
- Read the stack pointer and the memory it points to.

Part 2: Understanding Processor Registers

What is a Register?

A register is very fast storage inside the CPU. Instructions use registers for values, addresses, temporary results, and control flow.

The ARM Cortex-M33 Registers

Register	Also Called	Purpose
`r0` - `r12`	General purpose	Hold values and addresses while instructions run.
`r13`	SP	Points to the current top of the stack.
`r14`	LR	Holds the return address after a function call.
`r15`	PC	Points to the next instruction to execute.

General-Purpose Registers (`r0` - `r12`)

This lesson uses `r2`, `r3`, `r4`, `r6`, `r9`, and `r10`. The assembly saves their current values to SRAM, then restores them before the loop repeats.

The Stack Pointer (`r13` / SP)

The stack is a region of SRAM used for temporary values, saved registers, return addresses, and local variables. On Cortex-M, the standard stack grows toward lower addresses.

- A `push` lowers `sp` and writes values below the old stack pointer.
- A `pop` reads values at `sp` and raises `sp`.
- Each saved register occupies 4 bytes.

```
Higher addresses
+------------------+
| Old SP location  |
+------------------+
| Saved value      |
+------------------+
| Saved value      |  <- SP after a multi-register push
+------------------+
Lower addresses
```

The Link Register (`r14` / LR)

A `bl` instruction calls a function and stores the return address in `lr`. The compiler-generated prologue for `main` saves `lr` on the stack before calling `stdio_init_all`.

The Program Counter (`r15` / PC)

The Program Counter identifies the next instruction. In GDB, the `=>` marker in `disas main` points to the instruction that will run when you type `si`.

Part 3: Understanding Memory Layout

XIP - Execute In Place

The Pico 2 executes this firmware directly from external flash through XIP. The executable code normally begins at `0x10000000`.

Memory Map Overview

```
+-------------------------------------+
| Flash Memory (XIP)                  |
| Starts at: 0x10000000               |
| Contains: program instructions      |
+-------------------------------------+
| SRAM                                |
| Starts at: 0x20000000               |
| Contains: stack, heap, variables    |
+-------------------------------------+
```

Why the Stack Is in SRAM

The stack changes on every function call and return, so it must be writable. When GDB displays `$sp`, it should show an address in the `0x200...` SRAM range. `x/wx $sp` reads the 32-bit value currently at the top of that stack.

Part 3.5: Reviewing Our Stack Code

The file `0x0001a_stack.c` initializes standard I/O, then repeats this assembly block forever:

```
__asm volatile(
    "push {r4, lr}\n"
    "push {r3, r2, r6}\n"
    "stmdb sp!, {r9, r10}\n"
    "ldmia sp!, {r9, r10}\n"
    "pop {r2, r3, r6}\n"
    "pop {r4, lr}\n"
    ::: "memory");
```

Breaking Down the Code

First Push: `push {r4, lr}`

This lowers `sp` by 8 bytes. At the new stack pointer, the saved values are:

```
[sp + 4]      = lr    (higher address, visual top)
[sp]          = r4    (lower address)  <- SP <- TOP OF STACK
```

Second Push: `push {r3, r2, r6}`

This is deliberately written in a confusing order. The source says `r3` first, but an ARM register list is a set of registers, not an ordered sequence of operations. The assembler encodes the same register mask as `{r2, r3, r6}` and warns that the list is not ascending.

After one `si`, the stack layout proves the actual rule:

```
[sp + 16]     = lr    (highest address, visual top)
[sp + 12]     = r4
[sp + 8]      = r6
[sp + 4]      = r3
[sp]          = r2    (lowest address)  <- SP <- TOP OF STACK
```

The first three words were written by this instruction; `r4` and `lr` remain from the preceding `push {r4, lr}`. The lowest register number in this push is stored at the lowest address. Because the stack grows down, `[sp]` is the lower address and the older `lr` at `[sp + 16]` is the higher address at the visual top.

High Registers: `stmdb sp!, {r9, r10}`

The 16-bit Thumb `push` encoding cannot encode high registers `r8` through `r12`. `stmdb sp!` is the general full-descending stack instruction used for `r9` and `r10`.

```
[sp + 4]      = r10   (higher address, visual top)
[sp]          = r9    (lower address)  <- SP <- TOP OF STACK
```

The Restore Instructions

The next instructions restore the same groups in reverse stack-group order. This matters because the last group saved is at the current top of the stack:

```
ldmia sp!, {r9, r10}
pop {r2, r3, r6}
pop {r4, lr}
```

The stack pointer ends at the same value it had before the inline assembly, so the infinite loop does not consume stack space.

Compiling and Flashing to the Pico 2

Step 1: Compile the Code

From the project folder, build the program:

```
& "$env:USERPROFILE\.pico-sdk\ninja\v1.13.2\ninja.exe" -C build
```

The expected artifact is `build\0x0001a_stack.elf`. The assembler reports `register range not in ascending order` for the deliberately unordered source list. That warning is expected for this experiment.

Step 2: Flash and Verify

Use the Debug Probe to flash the ELF and compare target flash with the built image:

```
& "$env:USERPROFILE\.pico-sdk\openocd\0.12.0+dev\openocd.exe" -s "$env:USERPROFILE\.pico-sdk\openocd\0.1
```

You should see:

```
** Programming Finished **
** Verify Started **
** Verified OK **
** Resetting Target **
shutdown command invoked
```

`Verified OK` proves that the programmed bytes match the ELF. `shutdown command invoked` is normal because `exit` ends OpenOCD after flashing.

Part 4: Dynamic Analysis with GDB

Prerequisites

Before starting, you need:

1. A Pico 2 with the Debug Probe connected.
2. OpenOCD from the installed Pico SDK.
3. `arm-none-eabi-gdb`.
4. The verified `build\0x0001a_stack.elf` on the Pico 2.

Connecting to Your Pico 2 with OpenOCD

Open a terminal and start the debug server. Leave it running:

```
& "$env:USERPROFILE\.pico-sdk\openocd\0.12.0+dev\openocd.exe" -s "$env:USERPROFILE\.pico-sdk\openocd\0.1
```

OpenOCD listens for GDB connections on port `3333`.

VM Command

If the VM has `openocd` on its `PATH`, use the same command without the full executable path:

```
openocd -s "$env:USERPROFILE\.pico-sdk\openocd\0.12.0+dev\scripts" -f interface/cmsis-dap.cfg -f target/
```

The VM needs USB access to the Debug Probe. Nothing else about the build, ELF, or GDB sequence changes.

Connecting to Your Pico 2 with GDB

Open a second terminal in the project folder:

```
arm-none-eabi-gdb build\0x0001a_stack.elf
```

Connect, reset, halt, and stop at `main`:

```
target extended-remote :3333
monitor reset halt
b main
c
disas main
```

You should see this instruction pattern from the current `build\0x0001a_stack.elf`. The instruction addresses are determined by the ELF. The register contents shown later are live target state and are not determined by the ELF.

```
=> main:          push    {r3, lr}
   main+2:        bl      stdio_init_all
   main+6:        push    {r4, lr}
   main+8:        push    {r2, r3, r6}
   main+10:       stmdb   sp!, {r9, r10}
   main+14:       ldmia.w sp!, {r9, r10}
   main+18:       pop     {r2, r3, r6}
   main+20:       ldmia.w sp!, {r4, lr}
   main+24:       b.n     main+6
```

Notice that GDB displays `{r2, r3, r6}`, not the source spelling `{r3, r2, r6}`. That is the encoded register set in canonical order. Some disassemblers display `r10` as its conventional alias, `sl`; `{r9, sl}` and `{r9, r10}` name the same register set.

Basic GDB Commands: Your First Steps

Command	Short Form	What It Does
`break main`	`b main`	Set a breakpoint at `main`.
`continue`	`c`	Run until a breakpoint.
`disassemble`	`disas`	Show the current function's assembly.
`info registers`	`i r`	Display CPU registers.
`stepi`	`si`	Execute exactly one instruction.
`nexti`	`ni`	Execute one instruction without entering a call.
`x/wx ADDRESS`		Examine one 32-bit hexadecimal word.
`monitor reset halt`		Ask OpenOCD to reset and halt the target.

Watching the Stack Change

> **Important: the instruction addresses and stack offsets below come from the current ELF. The Step 4 register capture is from the live GDB session shown here. Register contents and old SRAM words are target state, not constants that can be recovered from the ELF. The deterministic rule is that `push {r4, lr}` decreases `sp` by 8 bytes, stores `r4` at the new `[sp]`, and stores `lr` at `[sp + 4]`.

Step 1: Inspect the Stack Before the Compiler Prologue

At the breakpoint, GDB is paused before `push {r3, lr}`. Inspect the current stack pointer and the two words below it:

```
(gdb) p/x $sp
$1 = 0x20082000
(gdb) x/2wx $sp-8
0x20081ff8:     0x88526891      0x10000187
```

The stack pointer is at `0x20082000`. The two words below it contain previous values (before the prologue).

Step 2: Execute One Instruction

Execute the compiler prologue `push {r3, lr}`:

```
(gdb) si
0x100001e2      7           stdio_init_all();
(gdb) p/x $sp
$2 = 0x20081ff8
(gdb) p/x $r3
$3 = 0xe000ed08
(gdb) x/wx $sp
0x20081ff8:     0xe000ed08
(gdb) p/x $lr
$4 = 0x1000018b
(gdb) x/wx $sp+4
0x20081ffc:     0x1000018b
```

The prologue saved two values to the stack:

```
[sp + 4]      = 0x1000018b  (lr, higher address, visual top)
[sp]          = 0xe000ed08  (r3, lower address)  <- SP <- TOP OF STACK
```

Stack layout after prologue (memory grows downward; **higher hex addresses = deeper in stack = older data**):

```
Address       Pointer      Value         Label
0x20082000   ----------   ----------    (previous data - HIGHEST address, visual top)
0x20081ffc   ----------   0x1000018b    (lr, higher address)
0x20081ff8   <- [sp]      0xe000ed08    (r3, lowest address)  <- SP <- TOP OF STACK

MEMORY ORDER:  0x20081ff8 < 0x20081ffc < 0x20082000
         (visual bottom)          (visual top)
```

Memory address explanation for newcomers:

- Hex address `0x20081ffc` is **LARGER** than `0x20081ff8` (compare the last hex digits: `ffc` > `ff8`)

- **Larger addresses = higher in memory = deeper in the stack**
- When we PUSH, sp **DECREASES** (goes to a smaller address, moving DOWN on the page)
- When we POP, sp **INCREASES** (goes to a larger address, moving UP on the page)

The stack pointer dropped 8 bytes: from `0x20082000` to `0x20081ff8` (it went DOWN, to a smaller/lower address).

Step 3: Step Over `stdio_init_all`

Do not step into the library initialization code. Use `ni` to execute it and return:

```
(gdb) ni
0x100001e6      15          __asm volatile(
(gdb) disas main
Dump of assembler code for function main:
   0x100001e0 <+0>:     push    {r3, lr}
   0x100001e2 <+2>:     bl      0x10001594 <stdio_init_all>
=> 0x100001e6 <+6>:     push    {r4, lr}
   0x100001e8 <+8>:     push    {r2, r3, r6}
   ...
```

The arrow now points at `push {r4, lr}`, the first inline assembly instruction.

Step 4: Prove the First Inline Push

Read the registers and stack pointer before the first inline push:

```
(gdb) p/x $r4
$3 = 0x100001cc
(gdb) p/x $lr
$4 = 0x1000159b
(gdb) p/x $sp
$5 = 0x20081ff8
```

Now execute the first inline push:

```
(gdb) si
0x100001e8      16          "push {r4, lr}\n"
(gdb) p/x $sp
$6 = 0x20081ff0
(gdb) x/wx $sp
0x20081ff0:     0x100001cc
(gdb) x/wx $sp+4
0x20081ff4:     0x1000159b
```

After first inline push `push {r4, lr}`:

- SP = `0x20081ff0` (**new TOP of stack, sp decreased**)
- `r4` = `0x100001cc` (now saved on stack)
- `lr` = `0x1000159b` (now saved on stack)

The first push saved:

```
[sp + 4]      = 0x1000159b  (lr, higher address, visual top)
[sp]          = 0x100001cc  (r4, lower address)  <- SP <- TOP OF STACK
```

Stack layout after first inline push:

```
Address       Pointer      Value         Label
0x20081ffc   ----------   0x1000018b    (lr from prologue - highest address, visual top)
0x20081ff8   ----------   0xe000ed08    (r3 from prologue - higher address)
0x20081ff4   ----------   0x1000159b    (lr - higher address)
0x20081ff0   <- [sp]      0x100001cc    (r4, lowest address)  <- SP <- TOP OF STACK

MEMORY ORDER:  0x20081ff0 < 0x20081ff4 < 0x20081ff8 < 0x20081ffc
       (visual bottom)                    (visual top)
       Most recent                         Least recent
```

The stack pointer dropped from `0x20081ff8` to `0x20081ff0` — that's 8 bytes down (toward lower addresses). The value at `[sp]` (the TOP) is now `0x100001cc` (r4). Address `0x20081ff0` is SMALLER than `0x20081ff8`, so sp moved DOWN.

Step 5: Prove Register-List Ordering

Read the three registers before executing the deliberately unordered list `push {r3, r2, r6}`:

```
(gdb) p/x $r2
$18 = 0x200005cc
(gdb) p/x $r3
$19 = 0xe000ed08
(gdb) p/x $r6
$20 = 0x04f54710
(gdb) p/x $sp
$21 = 0x20081ff0
```

Now execute the second inline push:

```
(gdb) si
0x100001ea <+10>:       stmdb   sp!, {r9, sl}
(gdb) p/x $sp
$22 = 0x20081fe4
(gdb) x/wx $sp
0x20081fe4:     0x200005cc
(gdb) x/wx $sp+4
0x20081fe8:     0xe000ed08
(gdb) x/wx $sp+8
0x20081fec:     0x04f54710
```

The second push saved three registers in **numeric order** despite the source spelling `{r3, r2, r6}`:

```
[sp + 8]      = 0x04f54710  (r6, higher address, visual top)
[sp + 4]      = 0xe000ed08  (r3, higher address)
[sp]          = 0x200005cc  (r2, lower address)  <- SP <- TOP OF STACK
```

Stack layout after second inline push (lower hex addresses = most recent = TOP):

```
Address       Pointer      Value         Label
0x20081ffc   ----------   0x1000018b    (lr - HIGHEST address, OLDEST data, visual top)
0x20081ff8   ----------   0xe000ed08    (r3 - higher address)
0x20081ff4   ----------   0x1000159b    (lr - higher address)
0x20081ff0   ----------   0x100001cc    (r4 - higher address)
0x20081fec   ----------   0x04f54710    (r6 - higher address)
0x20081fe8   ----------   0xe000ed08    (r3 - higher address)
0x20081fe4   <- [sp]      0x200005cc    (r2, lowest address)  <- SP <- TOP OF STACK

ADDRESS ORDERING:  0x20081fe4 < 0x20081fe8 < ... < 0x20081ffc
                  (visual bottom)                  (visual top)
                  Hex comparison: fe4 < fe8 < fec < ff0 < ff4 < ff8 < ffc
                  Most recent data at the visual bottom
```

The stack pointer dropped from `0x20081ff0` to `0x20081fe4` — that's 12 bytes down (three 4-byte registers). When comparing hex: `0x20081fe4` is SMALLER than `0x20081ff0`, so sp moved to a LOWER address. The value at `[sp]` (the TOP) is now `0x200005cc` (r2).

Notice: the source wrote `r3` first, but the CPU pushed `r2` first because `r2` has a lower register number. The encoded register mask is `{r2, r3, r6}`, and the stack layout proves it.

Step 6: Prove the High-Register Save

Read the high registers before the `stmdb sp!, {r9, r10}` instruction:

```
(gdb) p/x $r9
$23 = 0x00000000
(gdb) p/x $r10
$24 = 0x10000000
(gdb) p/x $sp
$25 = 0x20081fe4
```

Execute the high-register save:

```
(gdb) si
0x100001ee <+14>:       ldmia.w sp!, {r9, sl}
(gdb) p/x $sp
$26 = 0x20081fdc
(gdb) p/x $r9
$27 = 0x00000000
(gdb) p/x $r10
$28 = 0x10000000
(gdb) x/wx $sp
0x20081fdc:     0x00000000
(gdb) x/wx $sp+4
0x20081fe0:     0x10000000
(gdb) x/wx $sp+8
0x20081fe4:     0x200005cc
(gdb) x/wx $sp+12
0x20081fe8:     0xe000ed08
(gdb) x/wx $sp+16
0x20081fec:     0x04f54710
(gdb) x/wx $sp+20
0x20081ff0:     0x100001cc
(gdb) x/wx $sp+24
0x20081ff4:     0x1000159b
(gdb) x/wx $sp+28
0x20081ff8:     0xe000ed08
```

After high-register save `stmdb sp!, {r9, r10}`:

- SP = `0x20081fdc` (**new TOP of stack, sp decreased further**)
- `r9` = `0x00000000` (now saved on stack)
- `r10` = `0x10000000` (now saved on stack)

The `stmdb sp!, {r9, r10}` instruction saved:

```
[sp + 4]      = 0x10000000  (r10, higher address, visual top)
[sp]          = 0x00000000  (r9, lower address)  <- SP <- TOP OF STACK
```

The stack pointer dropped from `0x20081fe4` to `0x20081fdc` — that's 8 more bytes down. The value at `[sp]` (the TOP) is now `0x00000000` (r9). This is the DEEPEST point of the stack during inline assembly.

```
Address       Pointer      Value         Label
0x20081ff8   ----------   0xe000ed08    (saved prologue r3, highest address, visual top)
0x20081ff4   ----------   0x1000159b    (lr, higher address)
0x20081ff0   ----------   0x100001cc    (r4, higher address)
0x20081fec   ----------   0x04f54710    (r6, higher address)
0x20081fe8   ----------   0xe000ed08    (r3, higher address)
0x20081fe4   ----------   0x200005cc    (r2, higher address)
0x20081fe0   ----------   0x10000000    (r10, higher address)
0x20081fdc   <- [sp]      0x00000000    (r9, lowest address)  <- SP <- TOP OF STACK
```

Step 7: Watch the Restores

At this point, the stack is at maximum depth. Now execute the restore instructions one by one:

Restore 1: ldmia sp!, {r9, r10}

```
(gdb) si
0x100001f2 <+18>:       pop     {r2, r3, r6}
(gdb) p/x $sp
$29 = 0x20081fe4
(gdb) p/x $r9
$30 = 0x00000000
(gdb) p/x $r10
$31 = 0x10000000
```

`ldmia sp!` restored `r9` and `r10` from the stack and raised `sp` by 8 bytes (from `0x20081fdc` to `0x20081fe4`). **SP is now `0x20081fe4`; `[sp]` contains r2, so `SP <- TOP OF STACK` at the next group.**

Restore 2: pop {r2, r3, r6}

```
(gdb) si
0x100001f4 <+20>:       ldmia.w sp!, {r4, lr}
(gdb) p/x $sp
$32 = 0x20081ff0
(gdb) p/x $r2
$33 = 0x200005cc
(gdb) p/x $r3
$34 = 0xe000ed08
(gdb) p/x $r6
$35 = 0x04f54710
```

`pop {r2, r3, r6}` restored the three registers from SRAM and raised `sp` by 12 bytes (from `0x20081fe4` to `0x20081ff0`). **SP is now `0x20081ff0`; `[sp]` contains r4, so `SP <- TOP OF STACK` at the final group.**

Restore 3: ldmia.w sp!, {r4, lr}

```
(gdb) si
0x100001f8 <+24>:       b.n     0x100001e6 <main+6>
(gdb) p/x $sp
$36 = 0x20081ff8
(gdb) p/x $r4
$37 = 0x100001cc
(gdb) p/x $lr
$38 = 0x1000159b
```

`ldmia.w sp!, {r4, lr}` restored `r4` and `lr` from SRAM and raised `sp` by 8 bytes (from `0x20081ff0` to `0x20081ff8`). **SP is now `0x20081ff8`; `[sp]` contains the saved prologue r3, so `SP <- TOP OF STACK` for the remaining prologue stack data.**

Result: After all three restore instructions, `sp` is back at `0x20081ff8`, exactly where it was before the inline assembly block started. The loop branches back to `push {r4, lr}` at address `0x100001e6` and repeats forever without consuming stack space.

Understanding the Stack Diagram

At the deepest point, after `stmdb sp!, {r9, r10}`, the inline assembly has saved 28 bytes. The `old SP` below is the stack pointer after the separate compiler prologue, not the stack pointer at entry to `main`:

```
Before inline assembly:             At maximum inline stack depth:

old SP  <- SP                        old SP
                                     [old SP - 4]  lr
                                     [old SP - 8]  r4
                                     [old SP - 12] r6
                                     [old SP - 16] r3
                                     [old SP - 20] r2
                                     [old SP - 24] r10
                                     [old SP - 28] r9  <- SP
```

The restoration sequence removes the top group first: `r9/r10`, then `r2/r3/r6`, then `r4/lr`.

Part 5: Summary and Review

What We Learned

1. **Registers**: `sp`, `lr`, and `pc` control stack location, returns, and the next instruction.
2. **The stack**: It grows down in SRAM. Pushes decrease `sp`; pops increase it.
3. **Multi-register instructions**: Register-list source order is not the stack-memory order. ARM stores lower register numbers at lower addresses.
4. **OpenOCD and GDB**: OpenOCD connects to the Debug Probe; GDB connects to OpenOCD on port `3333`.
5. **Live evidence**: `si` executes one instruction, and `x/wx $sp` shows the exact 32-bit word that instruction placed at the stack pointer.

The Program Flow

```
+-----------------------------------------------------+
| 1. push {r3, lr}                                    |
|    Compiler saves its prologue registers            |
+-----------------------------------------------------+
| 2. bl stdio_init_all                                |
|    Initialize standard I/O                          |
+-----------------------------------------------------+
| 3. push {r4, lr}                                    |
|    Save the first inline group                      |
+-----------------------------------------------------+
| 4. push {r3, r2, r6}                                |
|    Source order differs from stack-memory order     |
+-----------------------------------------------------+
| 5. stmdb / ldmia / pop / pop                        |
|    Save and restore all groups                      |
+-----------------------------------------------------+
| 6. b.n main+6                                       |
|    Repeat with the original stack pointer           |
+-----------------------------------------------------+
```

Key Takeaways

1. **The stack grows downward**: a push decreases the numeric value in `sp`.
2. **A register list is not a sequence**: `{r3, r2, r6}` and `{r2, r3, r6}` encode the same register set.
3. **Memory is the proof**: after `si`, inspect `[sp]`, `[sp+4]`, and `[sp+8]` with GDB.
4. **The restore order matters**: always restore the most recently saved group first.
5. **Balanced stack operations are required**: the loop returns `sp` to its starting value on every iteration.

Glossary

Term	Definition
Assembly	Human-readable form of processor instructions.
Breakpoint	A debugger stop point.
Debug Probe	Hardware interface that lets OpenOCD communicate with the target over SWD.
GDB	GNU Debugger, used to inspect and control the running target.
LR	Link Register, holding a function return address.
OpenOCD	Debug server that bridges GDB and the Debug Probe.
PC	Program Counter, pointing to the next instruction.
SP	Stack Pointer, pointing to the top of the stack.
SRAM	Writable memory used for runtime data and the stack.
XIP	Execute In Place, executing program code directly from flash.