NSEC workshop 2025

Bare Metal emulation

2025-05-16

About me

Was system administrator at McGill Genome Center

Joined ETTIC team at Desjardins in 2016

Did infrastructure testing

Then application testing

And now doing vulnerability research in IOT devices.

Main objectives of the workshop

- Learn how the toolchain create an executable file
- Observe what a cpu does first, after power up
- Run our own code directly on the cpu as if we were developping a bootloader
- Do basic reconnaissance on machines emulated by qemu
- Break a NSEC grade implementation of firmware encryption

Schedule

- We start by building the docker container, that will take some time.
- While container is building, we review the theory and tools we'll need for the workshop.
- At this point, our docker container should be ready to run.
- We will do each exercise then go over the solution in sequence.

Building the docker container

The workshop has been tester with `Debian` `docker.io` package, should work with the current version.

If you need docker to go through a proxy, you may use the `config.json` and the `http-proxy.conf` files provided to configure it.

Building the container

```
docker build --rm --tag workshop2025 .
```

Running docker container

```
docker run -it --rm -p 8888:8888 workshop2025
```

Now a bit of theory

- CPU architectures and frequently used registers
- Registers you'll interact with
  - Program counter (PC)
  - stack pointer (SP)
- Reset vector
- Toolchain and linker script
  - C Runtime
- Memory mapped peripherals

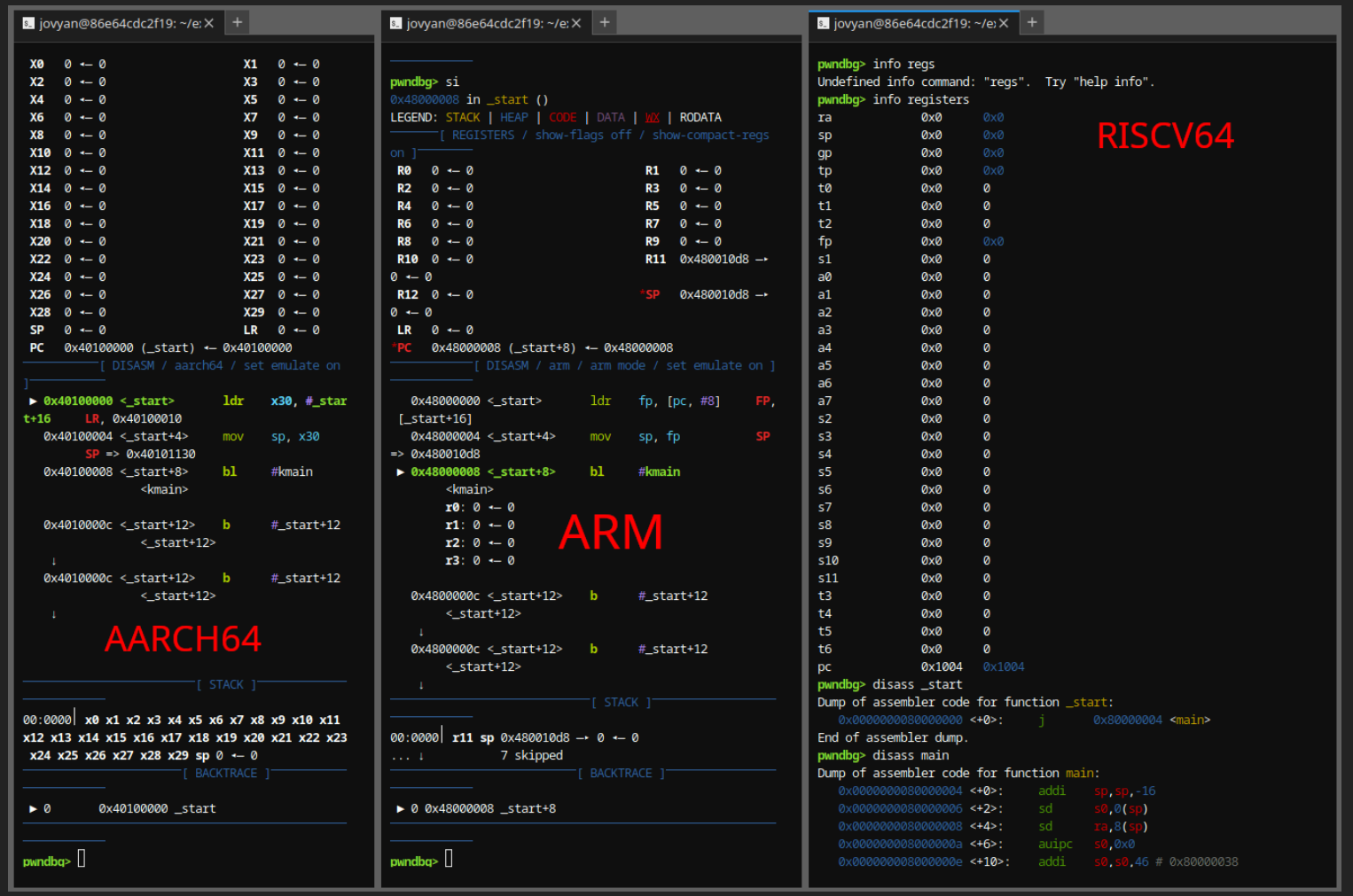
Architectures and registers

The architecture define the cpu itself and, among other things, the instructions we need to interact with it.

We will be using 3 different architectures:

- AARCH64 (ARM 64 bits)
- ARM 32 bits
- RISCV64 (RISCV 64 bits)

Here is a short assembly dump of code running on those CPU architectures.



From the previous image, we can notice similarities between architectures

In all cases:

- Program counter register also known as intruction pointer is called `PC`
- Stack pointer register is called `SP`
- Instructions are encoded in **mostly** fixed size, 16 or 32 bits, as opposed to `Intel` variable instruction size.

Reset vector

- CPU set instruction pointer at hardcoded address when `RESET` is activated e.g. at power up.
- Contains the very first instructions to be executed
- Reset vector usually calls `C runtime` so our code could run.
- Since the stack grows backward, the stack pointer is usually set high in the memory space.
- When facing an unknown firmware dump, try finding instructions that sets the stack pointer to an absolute value to locate the entry point.

Toolchain and linker script

The toolchain is the collection of programs we need to use to build an executable file that will run on our emulated CPU.

Among those programs, we have:

- The compiler: Turns our `C` code into object files which are just blob of compiled code for a specific architecture.
- The assembler: Turns assembly code into object files.
- The linker: Take all object files produced during compilation and assembly steps and construct the final executable according to instructions in the linker script.

Reference: https://bottomupcs.com/ch07.html

C Runtime

The `C Runtime` set the stage so our code can run.

Here is some of the things it needs to do:

- Initialize the stack by setting stack pointer.
- Prepare `argc` and `arvg` arguments for main() function
- Invoke constructors for global and static objects (in C++)

Look for `crt.s` or `boot.s` files to find code related to `C Runtime` in the exercises.

Since we are running on "bare metal", the `C Runtime` needs to run before our main function and at the very lease, set the stack pointer.

Therefore, it may run as the reset vector or be called by the reset vector like in the 3rd exercise.

So the call graph will look like this:

[Reset vector] -> [C Runtime] -> [Our entry point (main or start)]

Reference: https://medium.com/@inferara/understanding-the-c-runtime-crt0-crt1-crti-and-crtn-40e4b34f2a62

Memory mapped peripherials

- Peripherials are assigned memory addresses.
- A given peripherial like a `PL011` can be located at different memory adresses on different boards.
- Peripherials and their addresses can be listed using the `info mtree` command in `Qemu` monitor or by extracting the flattened device tree.

Flattened device tree extraction

The flattened device tree is a data structure which describe memory mapped peripherals and their location.

It may be used the the kernel, the bootloader or both to access peripherals on the machine.

It may be embedded in binary form in a firmware file.

Fortunately, it is possible to convert that blob back to human readable form.

The `device-tree-compiler` package provide the `dtc` utility to do so.

At last, `Qemu` offer the `dumptdb` argument to machine selection to extract the flattened device tree of the selected machine.

Here is an example for the `sifive_u` `RISCV` board

```
qemu-system-riscv64 -machine sifive_u,dumpdtb=sifive_u.dtb -nographic
dtc -I dtb -O dts sifive_u.dtb >> sifive_u.dts
```

The `UART` being used for stdout might even be marked in the resulting `dts` file

```
    chosen {
        stdout-path = "/soc/serial@10010000";
    };
    aliases {
        serial0 = "/soc/serial@10010000";
        serial1 = "/soc/serial@10011000";
        ethernet0 = "/soc/ethernet@10090000";
    };
```

See `Bootlin's` Device tree 101 for more information on flattened device trees: https://bootlin.com/pub/conferences/2021/webinar/petazzoni-device-tree-101/petazzoni-device-tree-101.pdf

About Qemu

We will be using `Qemu` full system emulation as opposed to user mode emulation.

- Qemu can emulate cpu of several architectures, boards and peripherals like uart, flash, busses like `USB` or `PCI`.
- We can load binaries to emulate using the -kernel or -device option.
- Qemu can also expose a `gdb` stub and stop execution until a `gdb` debugger connects to it and allow execution to proceed.

Running Qemu

First, we need to select appropriate `qemu` binary according to the architecture of the machine we want to emulate

- `qemu-system-aarch64` for ARM 64 bits machines (AARCH64). Note than it can also emulate ARM 32 bits machines
- `qemu-system-riscv64` for RISCV 64 bits machines.

Then, we need to select the specific machine we wish to emulate, to do so, we use the `-M` option

Using `-M help` will list all supported machines for the architecture we wish to emulate

As an example, the 10 first machines supported by `qemu-system-aarch64`

```
qemu-system-aarch64 -M help
Supported machines are:
ast1030-evb          Aspeed AST1030 MiniBMC (Cortex-M4)
ast2500-evb          Aspeed AST2500 EVB (ARM1176)
ast2600-evb          Aspeed AST2600 EVB (Cortex-A7)
ast2700-evb          Aspeed AST2700 A0 EVB (Cortex-A35) (alias of ast2700a0-evb)
ast2700a0-evb        Aspeed AST2700 A0 EVB (Cortex-A35)
ast2700a1-evb        Aspeed AST2700 A1 EVB (Cortex-A35)
b-l475e-iot01a       B-L475E-IOT01A Discovery Kit (Cortex-M4)
bletchley-bmc        Facebook Bletchley BMC (Cortex-A7)
bpim2u               Bananapi M2U (Cortex-A7)
canon-a1100          Canon PowerShot A1100 IS (ARM946)
...
```

Loading a binary for emulation in Qemu

Qemu offers several mecanisms to load and emulate software, we will be using a few of them:

- The `-kernel` option, like -kernel kernel.elf
- The `-device` option, like -device loader,file=hello,addr=0x80000000

The kernel option

The `-kernel` option will load an `ELF` file into memory and (usually) run it.

There is some cases when the `ELF` file won't run right away:

- We may want to use the `-kernel` option to load a `Linux` kernel but use the `-device` option to load a bootloader and pass control to it.
- The machine we wish to emulate will run hardcoded code before it get to run the code we specified. The `Zeroth stage bootloader` in `RISCV` is such a case we will cover.

Our `ELF` file will be parsed to be loaded and ran according to information in headers like the base address and entry point.

The device option

- The device option allows to load either an executable file with a known format like `ELF` or raw binary code
- The device option also allow to set the instruction pointer to the address where the file has been loaded.

Here is a simple example that will load the `hello` file at address **0x80000000**.

```
-device loader,file=hello,addr=0x80000000
```

Reference: https://www.qemu.org/docs/master/system/generic-loader.html

Exercise 1

Objectives

- Understand the toolchain by looking at how an executable file is being created and loaded into memory
- Test our Qemu emulator

Main source files:

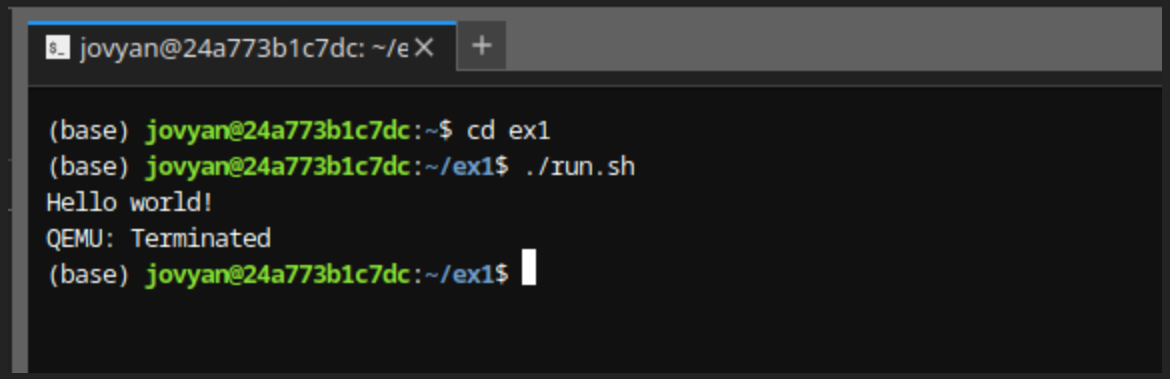
- Linker.ld Used to create elf file from object files created by compiler, define symbols used by boot.s (stack_top)
- boot.s Set stack pointer using the stack_top symbol, pass control to code defined in kernel.c by instruction `bl kmain`
- kernel.c Print "Hello world!" To do so, our code need to send the string "Hello world!", character by character to our emulated `UART`
- Makefile Contains instruction to compile and link everything and construct the final binary we will emulate using Qemu

Instructions:

- Use `make` command to compile, it should produce binary file `kernel.elf`
- Run `Qemu` as follows or simply use the `run.sh` script

```
qemu-system-aarch64 -machine virt -cpu cortex-a57 -kernel kernel.elf -nographic
```

You should see "Hello world!" printed on the terminal.



Use key combo `ctrl-a x` to terminate emulation and leave `Qemu`

Exercise 2

Objectives

- Learn how to use a debugger with Qemu
- Port `Hello world` code to a different machine

Instructions

The goal is to get the same code we used under exercise 1 but emulate it on a different machine and architecture
To do so, we need to compile the code to `ARM` 32 bits architecture using the provided Makefile then emulate it as follows

```
qemu-system-aarch64 -machine vexpress-a9  -cpu cortex-a9 -kernel kernel.elf -nographic
```

Does it print anything? If not then why?

Expose gdbstub with qemu

Use `Qemu` -gdb option to expose a gdbstub and -S option to stop execution at entry point.

Examples:

```
# To expose gdbstub on arbitrary port and wait for gdb to connect:
qemu-system-aarch64 -gdb tcp::1337 -S ...
# -s option shorthand for -gdb tcp::1234
qemu-system-aarch64 -s -S ...
```

Connect with gdb target remote

- Tell `gdb` to load the file: `file kernel.elf`
- If the file you are working on is not a knows executable file format, use the `set architecture` command to set architecture to match the one of the binary under Test
- Use the `set show-compact-regs on` to display a smaller set of register when using `pwngdb` under `gdb`
- At last, use the `target extended-remote` to connect to `Qemu`.

Example:

```
file kernel.elf
set show-compact-regs on
target extended-remote 127.0.0.1:1234
```

Debugging

- Use `si` (step into) command to execute one instruction. 'If you have `pwndbg` installed, you should see register and stack content
- Look for a pointer to our string, `hello world` being stored in one of the register.

Exercise 3

Objectives

- Understand the difference between `Qemu`'s options used to load code (-bios, -kernel and -device,loader=...)
- Trace the reset vector in Qemu, look it up in the source code
- Learn how to set the cpu state so the program could run (c runtime)
- Use reconnaissance techniques to locate the `UART` in the memory map
  - info mtree in `Qemu` monitor
  - flatten device tree extraction (useful on real hardware)

Instructions

We will be using code from the following repository:

- https://github.com/noteed/riscv-hello-c

It is already cloned in the `ex3` folder

Use the make command to build the `riscv` "Hello world!" binary then emulate as suggested in the `README.md` file.

```
qemu-system-riscv64 -nographic -machine sifive_u \
  -kernel build/bin/rv64imac/qemu-sifive_u/hello
```

You may also run the emulation from the `run.sh` file provided for convenience.

Memory overlapped error

At first, `Qemu` return the following error message:

```
/usr/local/share/qemu/opensbi-riscv64-generic-fw_dynamic.bin
 (addresses 0x0000000080000000 - 0x0000000080042878)
build/bin/rv64imac/qemu-sifive_u/hello ELF program header segment 1
 (addresses 0x0000000080000000 - 0x0000000080000040)
```

What is opensbi?

Do we want it?

If not, how to disable it?

CPU fan spinning but nothing on the console

Hint: What about the reset vector?

Try enabling the `gdb` stub with the `-s -S` command line options as we did in the last exercice

Can you explain why execution starts at address **0x1000** instead of the expected **0x80000000** ?

Can you find where the code in the reset vector in `Qemu` source code? Try searching in `hw/riscv/sifive_u.c`

Is the machine initialized properly? Can you find any register whose value might be problematic as program counter reaches **0x80000000**

Hello world still does not get printed

Hint: Try to find `UART` in the memory map

2 Methods:

- info mtree in `Qemu` monitor
- flattened device tree extraction

Is the `UART` is where our code expects it to be?

Exercise 4

CTF time!

Objective

- Break the firmware encryption implemented by a mock bootloader and get the flag.

Instructions

Here, our "bootloader" decrypt the root filesystem and would pass control to the kernel within it.

Then, how can we gain access to the unencrypted filesystem?

- Use static analysis with your favorite tool.
- Since our goal is to access the firmware in plaintext, can you locate any functions related to cryptography?
- At last, does the plaintext we seek resides in memory at some point?
- Could we extract it?

Solutions

Exercice 1

Using the `run.sh` script should start the emulator.

The code is meant to run out of the box, "Hello world!" should print on the terminal.

Exercise 2

Error in boot.s

The code responsible for setting the stack pointer and the frame pointer is using register only available in `AARCH64`. Since we are emulating an `ARM` 32 bits CPU, we need to find `ARM` 32 register used for frame pointer. Stack pointer is called `SP` in both cases.

```
make
arm-none-eabi-as boot.s -o boot.o
boot.s: Assembler messages:
boot.s:3: Error: ARM register expected -- `ldr x30,=stack_top'
boot.s:4: Error: immediate expression requires a # prefix -- `mov sp,x30'
make: *** [Makefile:6: all] Error 1
```

Under ARM 32 bits, frame pointer register is `R11`, also known as `FP`, so we change `X30` to `FP` in `boot.s`

Reference: https://en.wikibooks.org/wiki/Embedded_Systems/ARM_Microprocessors

Nothing happens, why?

Run `debug.sh` to enable the debugger, trace instructions in gdb.

```
pwndbg> file kernel.elf
pwndbg> set show-compact-regs on
pwndbg> target extended-remote 127.0.0.1:1234
Remote debugging using 127.0.0.1:1234
```

At the same time, open the monitor:

```
telnet localhost 55555
```

From the monitor, show memory map:

```
(qemu) info mtree
address-space: I/O
  0000000000000000-000000000000ffff (prio 0, i/o): io

address-space: cpu-memory-0
address-space: cpu-secure-memory-0
address-space: memory
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000000000-0000000003ffffff (prio 0, romd): alias vexpress.flashalias @vexpress.flash0 0000000000000000-0000000003ffffff
...
    0000000010009000-0000000010009fff (prio 0, i/o): pl011
    000000001000a000-000000001000afff (prio 0, i/o): pl011
    000000001000b000-000000001000bfff (prio 0, i/o): pl011
    000000001000c000-000000001000cfff (prio 0, i/o): pl011
...
    0000000040000000-0000000043ffffff (prio 0, romd): vexpress.flash0
    0000000044000000-0000000047ffffff (prio 0, romd): vexpress.flash1
    0000000048000000-0000000049ffffff (prio 0, ram): vexpress.sram
    000000004c000000-000000004c7fffff (prio 0, ram): vexpress.vram
    000000004e000000-000000004e0000ff (prio 0, i/o): lan9118-mmio
    0000000060000000-0000000067ffffff (prio 0, ram): vexpress.highmem
```

Compare the address in the instruction pointer (`PC`) with adresses, what do you see?

- Code is loaded in memory section marked as read-only, memory layout is different from the machine emulated in the first exercise.

Observer the behavior at `PC` = **0x40100058**, does the store instruction succeed?

- The `str` instruction fails as memory not writeable.

Appropriate memory address to load our code is the `sram` section for this particular machine. Modify the linker script, `linker.ld`, set the start address to **0x48000000** instead of **0x40100000**.

Try tracing instructions until you reach **0x48000058**, does the `str` instruction succeed?

You should see the address of the "Hello world!" string being written at location pointed by `FP`-8.

Memory is writeable but still nothing

Where is our `UART`?

Of course, peripherials are mapped to different memory location in different boards.

From our memory map, the first pl011 `UART` is mapped at **0x10009000**.

Therefore, `kernel.c` also needs to be modified.

```
volatile uint8_t *uart = (uint8_t *) 0x10009000;
```

Rebuild and run again, "Hello world!" should be displayed

Exercise 3

Qemu memory overlap error

```
/usr/local/share/qemu/opensbi-riscv64-generic-fw_dynamic.bin
 (addresses 0x0000000080000000 - 0x0000000080042878)
build/bin/rv64imac/qemu-sifive_u/hello ELF program header segment 1
 (addresses 0x0000000080000000 - 0x0000000080000040)
```

We need to disable opensbi to get rid of the memory overlap error

Disabling opensbi

Qemu tries to load opensbi at the same memory location our `kernel.elf` file is to be loaded, hence the overlap error.

One may add `-bios none` in qemu command line options to disable loading of opensbi

```
QEMUBIN=$HOME/src/qemu/build/qemu-system-riscv64

${QEMUBIN} -nographic -machine sifive_u \
  -bios none \
  -kernel build/bin/rv64imac/qemu-sifive_u/hello
```

What is opensbi

Opensbi runs with the highest level of CPU privilege, M-Mode. Responsible for authenticating and loading kernel, which may run at a lesser level of privileges (S-Mode)

References

- Riscv boot process: https://popovicu.com/posts/risc-v-sbi-and-full-boot-process/
- Riscv privilege levels (WD slide deck): https://riscv.org/wp-content/uploads/2024/12/13.30-RISCV_OpenSBI_Deep_Dive_v5.pdf

PC = 0x1000 at boot instead of 0x80000000

The Zeroth Stage Bootloader (ZSBL)

Once you power on this virtual machine, QEMU fills the memory at 0x1000 with a few instructions and sets the program counter right to that address.

0x1000 is referenced as the default reset vector, DEFAULT_RSTVEC = **0x00001000**

See `target/riscv/cpu.c` in qemu source code

```
// around line 1070
    env->pc = env->resetvec;
```

For the riscv `sifive_u` machine, the Zeroth stage boot loader is define in `hw/riscv/sifive_u.c` at line 620:

```
    /* reset vector */
    uint32_t reset_vec[12] = {
        s->msel,                          /* MSEL pin state */
        0x00000297,                       /* 1:  auipc  t0, %pcrel_hi(fw_dyn) */
        0x02c28613,                       /*     addi   a2, t0, %pcrel_lo(1b) */
        0xf1402573,                       /*     csrr   a0, mhartid  */
        0,
        0,
        0x00028067,                       /*     jr     t0 */
        start_addr,                       /* start: .dword */
        start_addr_hi32,
        fdt_load_addr,                    /* fdt_laddr: .dword */
        fdt_load_addr_hi32,
        0x00000000,
```

Below is a screen capture from a gdb session. Address 0x1000 contains the zero stage boot loader. The program counter will start at **0x1004**.

```
▶ 0x1004    auipc  t0, 0            T0 => 0x1004
  0x1008    addi   a2, t0, 0x2c     A2 => 0x1030 (0x1004 + 0x2c)
  0x100c    csrr   a0, mhartid
  0x1010    ld     a1, 0x20(t0)     A1, [0x1024] => 0x87e00000 ◂— 0xb130000edfe0dd0 // dtb
  0x1014    ld     t0, 0x18(t0)     T0, [0x101c] => 0x80000000 (_start) ◂— 0xe02211410040006f /* address of our code */
  0x1018    jr     t0                          <_start>
```

Data stored at **0x1000** is not executable code but the Mode Select (MSEL) which is data.

It is being used to set where the next stage will be loaded from. It can take the following values:

- 0x0: Defaut, next stage will be loaded from RAM
- 0x1: Will be loaded from memory mapped SPI flash (MSEL_MEMMAP_QSPI0_FLASH)
- 0x6: Will be loaded from SPI flash (MSEL_L2LIM_QSPI0_FLASH)
- 0x11: Will be loaded from SD card (MSEL_L2LIM_QSPI2_SD)

References

- Qemu source code: https://github.com/qemu/qemu/blob/master/hw/riscv/sifive_u.c
- Qemu headers : https://github.com/qemu/qemu/blob/master/include/hw/riscv/sifive_u.h
- Uros Popovic post: https://popovicu.com/posts/bare-metal-programming-risc-v/
- slavaim riscv notes: https://github.com/slavaim/riscv-notes/blob/master/bbl/boot.md

Stack pointer not set

Stack pointer does not get set by either ZSBL or our code.

We must set `SP` register to some sensible value. One possible value is top of RAM as stack will grow downwards.

First, we need to define the __stack_top symbol in the linker script

Our linker script is `env/qemu-sifive_u/default.lds`

Add a PROVIDE statement below the MEMORY STATEMENT to define the `__stack_top` symbol

```
PROVIDE(__stack_top = ORIGIN(ram) + LENGTH(ram));
```

Then, in the `C` runtime file, `env/qemu-sifive_u/crt.s`, add an instruction to load the `__stack_top` symbol into the `SP` register. That will set the stack pointer to the top of RAM.

```
_start:
    la      sp, __stack_top
```

Wrong UART address

The string "Hello World!" still does not get printed. The error message "<Cannot dereference [0x10013000]>" is show in gdb as we try to print a character.

The address **0x10013000** is supposed to be the location of our memory mapped `UART`, what went wrong?

Memory map reconnaissance: monitor method

Uart address is visible from memory tree in monitor

To access the monitor, one needs to enable it on the `Qemu` command line:

Note: It is recommanded to also enable the `gdb` stub so execution won't start right away

```
QEMUBIN=$HOME/src/qemu/build/qemu-system-riscv64

${QEMUBIN} -nographic -machine sifive_u \
  -bios none \
  -kernel build/bin/rv64imac/qemu-sifive_u/hello \
  -s -S \
  -monitor telnet:127.0.0.1:55555,server,nowait
```

Then, use netcat or similar tool to access it once `Qemu` is running

```
telnet localhost 55555
```

Use `info mtree` command to display the memory map and locate the `UART` address.

```
(qemu) info mtree
...
    0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000001000-000000000000ffff (prio 0, rom): riscv.sifive.u.mrom
    0000000002000000-000000000200ffff (prio 0, i/o): riscv.aclint.swi
    0000000002004000-000000000200bfff (prio 0, i/o): riscv.aclint.mtimer
    0000000002010000-000000000201ffff (prio -1000, i/o): riscv.sifive.u.l2cc
    0000000003000000-00000000030fffff (prio 0, i/o): sifive.pdma
    0000000008000000-0000000009ffffff (prio 0, ram): riscv.sifive.u.l2lim
    000000000c000000-000000000fffffff (prio 0, i/o): riscv.sifive.plic
    0000000010000000-000000001000ffff (prio 0, i/o): riscv.sifive.u.prci
    0000000010010000-000000001001001f (prio 0, i/o): riscv.sifive.uart <=== Our UART!!
    0000000010011000-000000001001101f (prio 0, i/o): riscv.sifive.uart
...
    0000000010090000-00000000100907ff (prio 0, i/o): enet
    00000000100a0000-00000000100a0fff (prio -1000, i/o): riscv.sifive.u.gem-mgmt
    00000000100b0000-00000000100bffff (prio -1000, i/o): riscv.sifive.u.dmc
    0000000020000000-000000002fffffff (prio 0, ram): riscv.sifive.u.flash0
    0000000080000000-0000000087ffffff (prio 0, ram): riscv.sifive.u.ram
```

Memory map reconnaissance: flattened device tree method

One may dump the device tree blob (dtb) using `Qemu` and convert it to source form (dts)

```
qemu-system-riscv64 -machine sifive_u,dumpdtb=sifive_u.dtb -nographic
dtc -I dtb -O dts sifive_u.dtb >> sifive_u.dts
```

From the device tree in source form (dts), our `UART` is easy to find

```
chosen {
    stdout-path = "/soc/serial@10010000";  <=== Our UART!!
};

aliases {
    serial0 = "/soc/serial@10010000";
    serial1 = "/soc/serial@10011000";
    ethernet0 = "/soc/ethernet@10090000";
};
```

Once you get the `UART` location using either method, update the file `libfemto/std/putchar.c`.

Set `UART` address as follows:

```
static volatile int *uart = (int *)(void *)0x10010000;
```

Rebuild and run to see "Hello World!" being printed.

Exercise 4

Capture the flag challenge

The objective is to recover the firmware in plaintext.

To succeed, we need either:

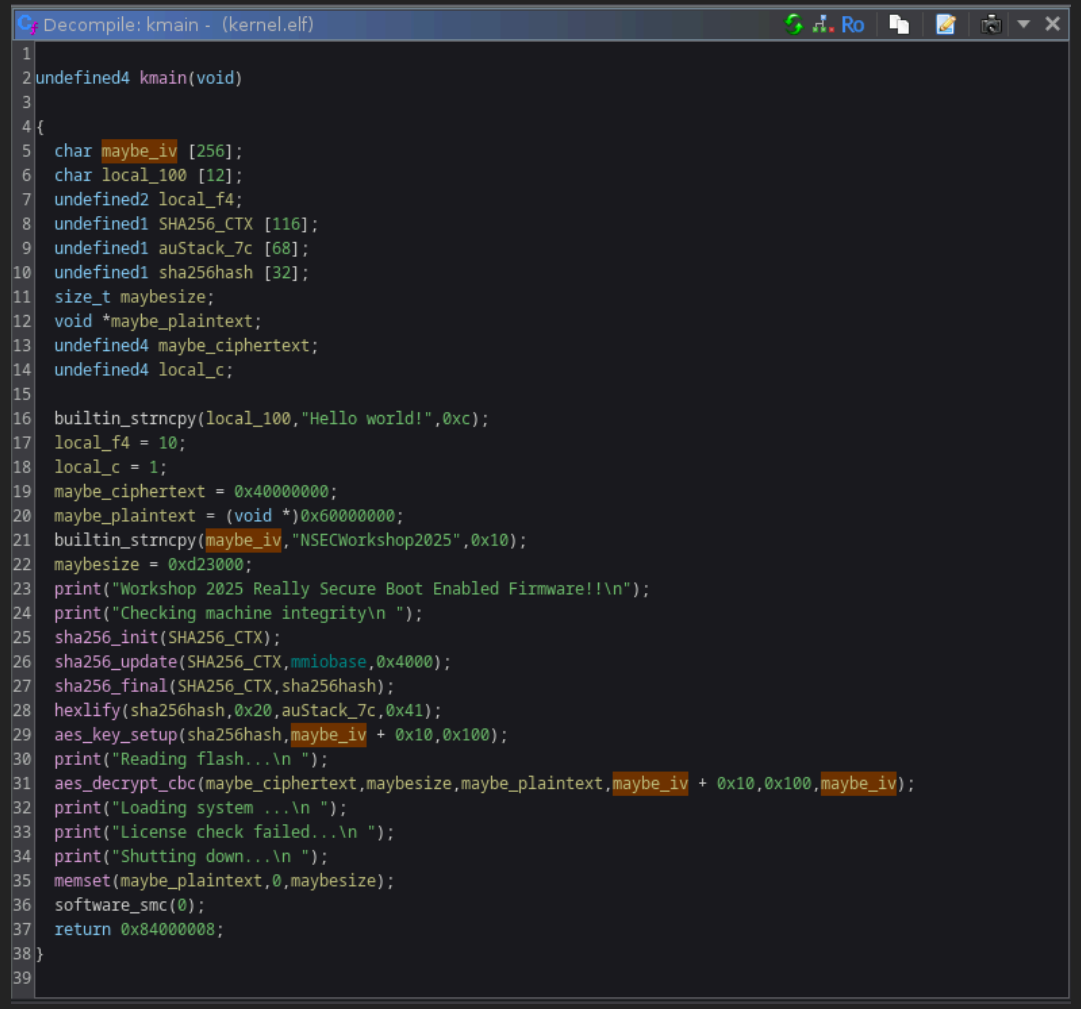
- The key
- The address of the memory location where plaintext is stored

Static analysis

Load `kernel.elf` in `Ghidra`, open it in code browser and analyse it.

Some suggestions:

- Define a `char[16]` at address **0x48006a58**, we want to figure out the purpose of that data
- Binary is not stripped, try to look up function names. We could afford to make one attempt assuming than these functions behave like those bearing the same name in known libraries. It not then it means we need to investigate further.
- If you can't find a function's documentation by looking up the name, try looking into it. Which arguments are being read? Which arguments are being written to?



Ghidra screen capture of static analysis

Dynamic analysis

Time to check hypotheses from static analysis

Set breakpoint at the print("Loading system ...") function.

Inspect the memory using arguments passed to aes_decrypt_cbc as pointers

Can you find the plaintext?

Use `Qemu` monitor to dump memory

```
# From Qemu monitor's help on memsave command
# memsave addr size file
memsave 0x60000000 0xd23000 plaintext.bin
```

One may also use `gdb` to dump the memory once the break point is reached.

Look at `dump memory` function in `gdb.

Plaintext analysis

Firmware root filesystem type still to be identified

Basic tools:

- file
- binwalk

Once you know which filesystem it is, you may access it and recover the flag.