# Attach a screenshot of you successfully running the given code

i.e. a screenshot of your emulator/rpi3 terminal printing "hello world".

A screen shot of a computer code

Description automatically generated

# First, inspect the kernel binary (kernel8.elf) of p1exp1.

You may use command line tools (e.g. objdump or readelf) or GUI tools (e.g. ODA). Btw, we have provided instructions <https://fxlin.github.io/p1-kernel/dump/>

**How many sections are in the elf file? What are these sections?**

Include your command line output as text (if you use command line tools) or attach screenshots (if you use GUI tools).

14 sections

1. <blanks>
2. .text.boot
3. .text
4. .rodata
5. .eh\_frame
6. .debug\_info
7. .debug\_abbrev
8. .debug\_aranges
9. .debug\_line
10. .debug\_str
11. .comment
12. .symtab
13. .strtab
14. .shstrtab

tkj9ep@granger1:~/p1-kernel/src/exp1$ readelf -S build/kernel8.elf

There are 14 section headers, starting at offset 0x10f60:

Section Headers:

[Nr] Name Type Address Offset

Size EntSize Flags Link Info Align

[ 0] NULL 0000000000000000 00000000

0000000000000000 0000000000000000 0 0 0

[ 1] .text.boot PROGBITS 0000000000080000 00010000

0000000000000030 0000000000000000 AX 0 0 4

[ 2] .text PROGBITS 0000000000080030 00010030

0000000000000250 0000000000000000 AX 0 0 4

[ 3] .rodata PROGBITS 0000000000080280 00010280

0000000000000010 0000000000000000 A 0 0 8

[ 4] .eh\_frame PROGBITS 0000000000080290 00010290

00000000000000b0 0000000000000000 A 0 0 8

[ 5] .debug\_info PROGBITS 0000000000000000 00010340

00000000000001dc 0000000000000000 0 0 1

[ 6] .debug\_abbrev PROGBITS 0000000000000000 0001051c

000000000000012b 0000000000000000 0 0 1

[ 7] .debug\_aranges PROGBITS 0000000000000000 00010650

00000000000000f0 0000000000000000 0 0 16

[ 8] .debug\_line PROGBITS 0000000000000000 00010740

00000000000001cc 0000000000000000 0 0 1

[ 9] .debug\_str PROGBITS 0000000000000000 0001090c

0000000000000145 0000000000000001 MS 0 0 1

[10] .comment PROGBITS 0000000000000000 00010a51

000000000000002b 0000000000000001 MS 0 0 1

[11] .symtab SYMTAB 0000000000000000 00010a80

0000000000000390 0000000000000018 12 26 8

[12] .strtab STRTAB 0000000000000000 00010e10

00000000000000c7 0000000000000000 0 0 1

[13] .shstrtab STRTAB 0000000000000000 00010ed7

0000000000000087 0000000000000000 0 0 1

**How many symbols are in the elf file?**

22 symbols; used nm command

**What is the address of symbol kernel\_main? What are the first 8 bytes at the symbol? What are the corresponding instructions?**

**Address**: 000000000008022c

**First 8 Bytes**: a9bf7bfd 910003fd

**Instructions**:

stp x29, x30, [sp, #-16]!

mov x29, sp

bl 0x80118 <uart\_init>

adrp x0, 0x80000 <\_start>

add x0, x0, #0x280

bl 0x800bc <uart\_send\_string>

bl 0x8007c <uart\_recv>

and w0, w0, #0xff

bl 0x80030 <uart\_send>

b 0x80244 <kernel\_main+24>

**How many bytes does each aarch64 instruction contain?**

Each aarch64 contains 4 bytes

**Now examine kernel8.img (use the hexdump command or the** [**VSCode plugin**](https://marketplace.visualstudio.com/items?itemName=slevesque.vscode-hexdump)**). Search for the first 8 bytes of kernel\_main(). Can you find it? At which offset of kernel8.img?**

I found the first 8 bytes of kernel\_main in memory address 0x22c

**How is kernel8.img generated out of kernel8.elf?**

The image file is generated by the ELF file by simply taking all the sections of the ELF file and combining them together into an image of which the machine should be able to run. In other words, the kernal8.elf is converted into a binary executable file that can be run by the machine.

# Second, some ARM64 exercise:

**How many general-purpose registers in aarch64 (i.e. the 64-bit execution state of ARM64)? How many bytes in each register?**

There are 31 general purpose registers, x0 – x30 where each register are of size 64-bit or 8 bytes.

**Use your own words, explain the following instructions, each in one short sentence.**

**and**: performs the logical AND operation between two values

**bl**: does a unconditional branch to a target address and stores the return address to the general purpose register x30; the ret instruction can be used to branch back to the target address

**mov**: moves a constant or a register value to another register

**adr**: grabs the relative address of a label and loads it into a target register

**mrs**: takes a value from the system register and loads it into one of the general purpose registers, that is x0 – x30

**Back to kernel8.elf. Use your own words, explain instruction by instruction: how the delay() function works.**



**subs x0, x0, #0x1**: decrements value in register x0 by 1; x0 = x0 – 1

**b.ne 0x00080260 <delay>**: if value in x0 is not equal to value in 0x00080260, then jump to address where delay function begins

**ret**: (if value in x0 is equal to value in 0x00080260) jump back to original caller of the function; address of original caller stored in register x30

# Third, about Rpi3

Watch the video [Eben Upton on Rpi3](https://youtu.be/37_7arZZlUI) and answer the questions:

What would be the benefit of supporting 64-bit (AArch64)? In particular, why it is important to support 64-bit by an operating system?

* Broader range of operating systems such as red hat
* Faster by 10 times
* More possible projects especially in IoT

Have you ever used Raspberry in any chance? What was for?

* I have not used a Raspberry before.