



POLITECNICO DI TORINO

Collegio di Elettronica, Telecomunicazioni e Fisica

Open Access Laboratory Experience for Introduction to Development of Linux Device Drivers for Emulated Cryptographic Cores with RISC-V ISA

Operating Systems

Master Degree in Electronic Engineering

Group 08

Shahabuddin Danish, Areeb Ahmed

Table of Contents

Lab 0: Setup of Lab Environment.....	2
Introduction.....	2
Installing and setting up Linux.....	2
Post-Installation Setup and System Updates.....	2
Installing GCC, Git and QEMU Dependencies.....	3
Installing QEMU.....	4
Installing Buildroot, Building the Bootloader and Target Linux Kernel.....	5
Lab 1: Understanding the SHA256 Algorithm.....	9
Exercise 1: Study the SHA256 Implementation.....	9
Exercise 2: Interactive SHA256 Algorithm Visualization.....	11
Lab 2: Exporting a custom cryptocore to our QEMU RISC-V machine.....	12
Exercise 1: Understanding the Cryptocore Model in QOM.....	12
Exercise 2: Registering the cryptocore as a QEMU Device.....	13
Lab 3: Device Driver Development and Configuration.....	20
Exercise 1: Developing and Loading a Simple Linux Kernel Module.....	20
Exercise 2: Compiling and loading the Cryptocore Device Driver.....	23
Lab 4: Cross-compilation for RISC-V.....	25
Exercise 1: Cross-compilation toolchains and Buildroot Cross-compiler.....	25
Exercise 2: Setting up the RISC-V Cross-compiler.....	26
Exercise 3: Cross-compilation of a simple program and running in Buildroot.....	27
Exercise 4: Cryptocore device driver cross-compilation and insertion.....	30
Lab 5: Testing SHA256 Cryptocore and Device Driver through Userspace.....	33
Exercise 1: Userspace Program to Read Cryptocore ID.....	33
Exercise 2: Userspace Program for SHA256 Hash Computation.....	34
Conclusion.....	35

Lab 0: Setup of Lab Environment

Introduction

This practical laboratory series focuses on a SHA256 cryptographic core in a simulated environment (QEMU) for the RISC-V instruction set architecture. This first lab serves as the foundation for understanding and utilizing the tools necessary for cryptographic hardware simulation. The primary goal for lab 0 is to set up, and get acquainted with the necessary tools and configurations, which are necessary to simulate a RISC-V machine on your PC for the upcoming exercises.

In this lab, you will install set up a suitable Linux distribution that supports the development and testing of a Linux device driver intended to interact with our modeled cryptographic core, and install and configure QEMU which is an emulator that will simulate the RISC-V machine that will host our cryptographic accelerator core.

Installing and setting up Linux

The objective of this exercise is to choose and install a Linux distribution that supports the necessary development tools and environment for running the simulation environment and testing the SHA256 cryptographic core. While most modern Linux distributions are suitable for our purposes, we will choose to go with Ubuntu for this lab experience as it is the most common and a popular choice in educational settings.

Visit the Ubuntu website to download the latest LTS version of Ubuntu Desktop. It is recommended to install the operating system on your PC in a dual-boot configuration to ensure optimal performance, which is also the reference for this lab experience. However, if that is not a possibility, using a virtual machine on your existing operating system is a viable alternative. In either case, it is recommended to reserve at least 50GB of disk space to avoid running into problems in the later labs.

For the VM option, you can download and install Oracle VM VirtualBox. Create your virtual machine, choosing the Ubuntu ISO as the installer disc image, and make sure to allocate at least 4GB of memory and as many CPU cores as possible to ensure the lab exercises run smoothly.

Post-Installation Setup and System Updates

After installation, boot into Ubuntu and execute the following command in the terminal to verify your installed version:

```
lsb_release -a
```

This will print distribution related information of your OS:

```
No LSB modules are available.  
Distributor ID: Ubuntu  
Description: Ubuntu 22.04.4 LTS  
Release: 22.04  
Codename: jammy
```

After this, we need to make sure the Ubuntu package repository is updated. To do this run the following commands,

```
sudo apt update  
sudo apt upgrade
```

Installing GCC, Git and QEMU Dependencies

Our first task will be to install the GCC compiler and its other utilities. To do this in Ubuntu you need to run the following command,

```
sudo apt install build-essential
```

This should install gcc, which you can also check using,

```
gcc --version
```

Running this command should give you an output as seen below. The current gcc version at the time of creating this lab experience is 11.4.0.

```
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0  
Copyright (C) 2021 Free Software Foundation, Inc.  
This is free software; see the source for copying conditions. There is NO  
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

The next thing to do is to install Git, which will be useful for sourcing QEMU. To do this we can run,

```
sudo apt install git
```

The git version can also be seen as we saw with gcc. In fact, every package we install can be checked similarly by using the package name followed by the `--version` argument. Now, finally we can move on to installing dependencies for building QEMU.

```
sudo apt-get install python3-sphinx
sudo apt-get install ninja-build
sudo apt-get install python3-sphinx-rtd-theme
sudo apt-get install meson
sudo apt-get install libglib2.0-dev
sudo apt-get install libpixman-1-dev
sudo apt-get install flex
sudo apt-get install bison
sudo apt-get install python3-venv
sudo apt-get install libslirp-dev
```

These packages include all the necessary build packages, compilation toolchain, graphical library, network library, etc. While building QEMU later, you might also see some runtime packages missing, but they are not necessary to install for our purpose.

Installing QEMU

We can now move on to the installation of QEMU. The installation process is straightforward and QEMU can be downloaded either through its website or through git. Here, we will be doing it through Git. To do this, create a new directory, and inside it run the following commands,

```
git clone https://gitlab.com/qemu-project/qemu.git
cd qemu
git submodule init
git submodule update --recursive
```

This fetches and clones the QEMU master repository. Now we need to invoke the configure script, which is a shell script that performs a lot of important tasks, but the main ones are as follows:

- It detects the architecture of the host machine.
- It lists targets for which to build emulators and which firmware binaries and tests to build.

- It invokes Meson in the virtual environment, to perform the actual configuration step for the emulator build.

Since we are going to simulate a cryptocore on a RISC-V machine, it is important that at this step we have to configure the QEMU build system to build the emulator for RISC-V. QEMU makes this process easier as it already has a 64-bit RISC-V machine build configuration defined, and we can build the emulator for that by passing the argument to the configure script. To do this we run,

```
./configure --target-list=riscv64-softmmu --enable-slrp  
make
```

make will invoke the makefile to compile the source code and perform the emulator build to build the QEMU executable using our specified configuration through the configure script. This process can take around 20-30 minutes, based on the performance of your system. Now whenever QEMU needs to be recompiled, it doesn't need to be reconfigured and we can just run make.

Installing Buildroot, Building the Bootloader and Target Linux Kernel

By now, we have built our target RISC-V machine in QEMU. But this is still a baremetal system for which we now need a bootloader, a cross-compiled Linux kernel and the root filesystem. For this purpose, we utilize Buildroot, a simple tool that can automate this process and build a complete Linux system for our QEMU machine. To do this, we now move to a separate directory and run the following commands,

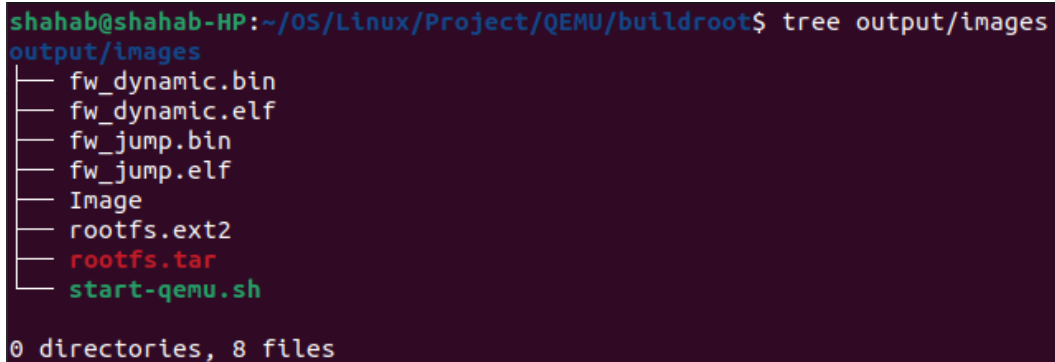
```
git clone https://github.com/buildroot/buildroot.git  
cd buildroot  
make qemu_riscv64_virt_defconfig  
make
```

These commands clone the Buildroot repository on your host machine (your PC), and then build the Buildroot Linux system for the virtual QEMU target machine. It conveniently includes the configuration for the virtual 64-bit RISC-V QEMU machine already defined with which we can configure and build Buildroot. It invokes the internal toolchain of Buildroot for cross-compiling a Linux kernel with a minimal rootfs for our target RISC-V machine with the default bootloader, also called the init system, BusyBox, which is a simple and basic init program without any advanced features but sufficient for our purposes.

Using Buildroot saves us from handling the init program and cross-compiling the Linux kernel for RISC-V by ourselves. Upon completion, the kernel image and the root filesystem can be seen inside the `buildroot/output/images` directory. We can look at it by running the following command in the `buildroot` directory,

```
tree output/images
```

Upon running this, we can see the linux kernel image and the root filesystem.



```
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot$ tree output/images
output/images
├── fw_dynamic.bin
├── fw_dynamic.elf
├── fw_jump.bin
├── fw_jump.elf
├── Image
├── rootfs.ext2
├── rootfs.tar
└── start-qemu.sh

0 directories, 8 files
```

You should note that here we also have a `start-qemu.sh` script. Buildroot provides this to easily boot up your RISC-V machine through the emulator, however it needs some modification before we can finally run our QEMU machine we built in the previous exercise.

At the end of the script you can find the following command,

```
exec <path_to_your_qemu>/build/qemu-system-riscv64 -M virt -bios
fw_jump.elf -kernel Image -append "rootwait root=/dev/vda ro" -drive
file=rootfs.ext2,format=raw,id=hd0 -device virtio-blk-device,drive=hd0
-netdev user,id=net0 -device virtio-net-device,netdev=net0 -nographic
${EXTRA_ARGS} "$@"
```

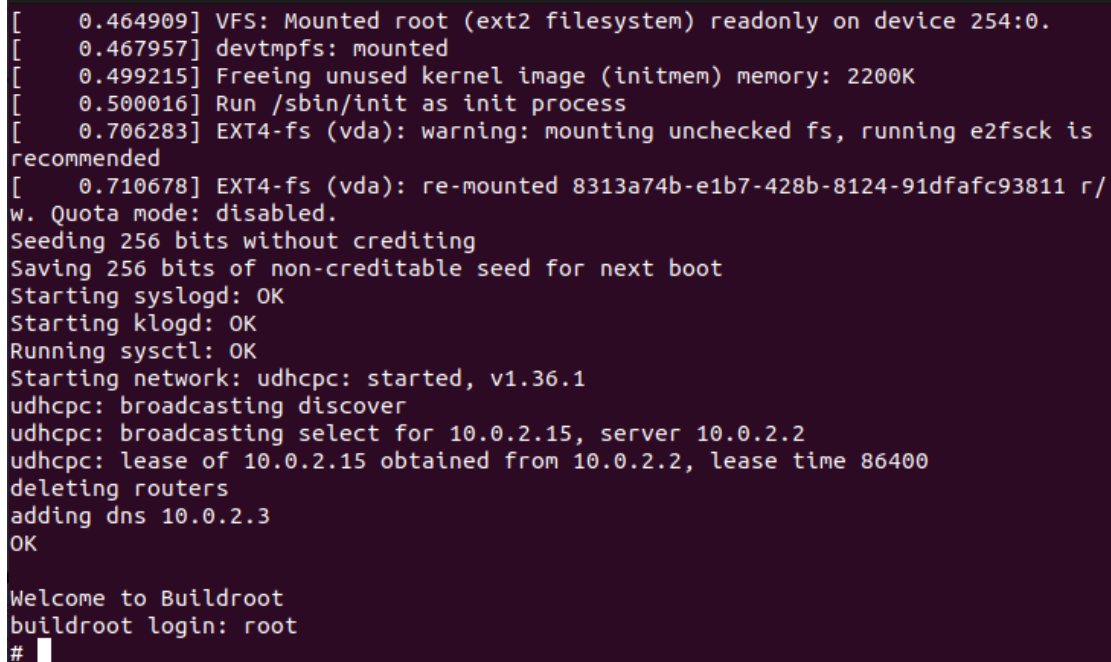
This command needs to point to the path of the QEMU executable binary that we built in the previous exercise. Therefore, you need to replace `<path_to_your_qemu>` with the complete path. For example in my case it looks like this,

```
exec ../../../../qemu/build/qemu-system-riscv64 -M virt -bios fw_jump.elf
-kernel Image -append "rootwait root=/dev/vda ro" -drive
file=rootfs.ext2,format=raw,id=hd0 -device virtio-blk-device,drive=hd0
-netdev user,id=net0 -device virtio-net-device,netdev=net0 -nographic
${EXTRA_ARGS} "$@"
```

We can now finally boot into our QEMU virtual RISC-V machine with Linux installed by invoking this script. To do that you can move into the `output/images` directory and run,

```
./start-qemu.sh
```

After running the script, the RISC-V emulator will boot up for the first time, providing details about the virtual machine, and the init program login prompt at the end. You can login to your virtual machine as the default login is **root**. This default login can be changed at the buildroot configuration stage, but it is not important for us to explore it now, however it is present in the buildroot documentation.



```
[ 0.464909] VFS: Mounted root (ext2 filesystem) readonly on device 254:0.
[ 0.467957] devtmpfs: mounted
[ 0.499215] Freeing unused kernel image (initmem) memory: 2200K
[ 0.500016] Run /sbin/init as init process
[ 0.706283] EXT4-fs (vda): warning: mounting unchecked fs, running e2fsck is
recommended
[ 0.710678] EXT4-fs (vda): re-mounted 8313a74b-e1b7-428b-8124-91dfafc93811 r/
w. Quota mode: disabled.
Seeding 256 bits without crediting
Saving 256 bits of non-creditable seed for next boot
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting network: udhcpd: started, v1.36.1
udhcpd: broadcasting discover
udhcpd: broadcasting select for 10.0.2.15, server 10.0.2.2
udhcpd: lease of 10.0.2.15 obtained from 10.0.2.2, lease time 86400
deleting routers
adding dns 10.0.2.3
OK

Welcome to Buildroot
buildroot login: root
#
```

To check everything is set up correctly, we can use the UNIX `uname` command to verify the kernel name, release number, version and the target machine architecture. To view all this information, you need to run,

```
uname --all
```

This will output all the necessary OS and hardware information for your Linux machine. For example, when run on my Buildroot target machine, I get the following output,


```
# uname --all
Linux buildroot 6.6.18 #1 SMP Sat May 11 19:21:51 CEST 2024 riscv64 GNU/Linux
#
```

This tells me I am running Linux, kernel version 6.6.18 on a riscv64 machine, where the current login username is buildroot. In this way we can verify that everything is set up correctly as we want it to. Similarly, if I run the same command in a different terminal on my host machine, I should get a different output.

```
shahab@shahab-HP:~$ uname --all
Linux shahab-HP 6.2.0-36-generic #37~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Mon Oct
 9 15:34:04 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
shahab@shahab-HP:~$
```

Here, as you can see I am still running Linux, but this time the username, kernel version and hardware architecture are all different since my host machine is running on x86_64 ISA with a different Linux kernel version 6.2.0.

These exercises finish lab 0, completing the environment setup that will be utilized in the next labs.

Lab 1: Understanding the SHA256 Algorithm

In this lab, you will take a look at the SHA256 cryptographic algorithm, a widely used hashing function that forms a crucial part of many security protocols and systems. It is important to understand how SHA256 works to be able to effectively understand how you can develop and troubleshoot cryptographic applications, including the QEMU cryptcore and the Linux device driver you will be working with in later labs.

Exercise 1: Study the SHA256 Implementation

The Secure Hash Algorithm 256-bit or SHA256 is part of the SHA2-2 family, designed to provide high collision resistance with its 256-bit output, which essentially means that it makes it very difficult to find two input strings that produce the same output hash. Since the SHA256 algorithm makes its outputs have 2^{256} possible hashes, it makes it computationally infeasible to be able to deliberately find a collision. However, this also makes the algorithm a bit tricky and confusing, which makes it important to be able to understand and visualize how it works.

In a nutshell, an input string goes through the following steps to compute the final hash, which in this lab exercise we will later visualize through an online interactive SHA256 calculator.

1. Preprocessing and Parsing

- a. After taking an input string, a message block is created in binary using the input string and immediately after the end of the string, a single bit "1" is added.
- b. At the end of the message block in the last 64 bits, the original string length is appended in binary in big-endian.
- c. The initial bit "1" is padded with zeros before the last 64 bits so that the message block becomes a multiple of 512 bits. If the input can't fit in a single message block, another 512 bits block is added and so on.

2. Message Schedule Preparation

- a. Each 512-bit block is broken into sixteen 32-bit words or chunks.
- b. These words are then extended into a 64-word message schedule array ($W[0..63]$) of 32 bits.
- c. The first chunk is copied into the first 16 words of the array ($W[0..15]$).
- d. The remaining array is computed using a series of logical functions that include XOR, right shifts, and rotations.

3. Compression

- a. Eight hash values from h0 to h7 are initialized, which are the first 32 bits of the fractional parts of the square roots of the first 8 prime numbers. These initial values are constants and are used in the compression function.
- b. Eight K constants are initialized, which are the first 32 bits of the fractional parts of the cube roots of the first 64 primes. These are also constants.
- c. Eight working variables (a, b, c, d, e, f, g, h) are initialized with the current hash value.
- d. Now there are 64 rounds of processing for each block. In each round, both the message schedule array and a set of predefined constants are used. Each round updates the working variables based on a complex mix of bitwise operations including shifts, rotations, modular additions, and the use of logical functions.
- e. At the end of processing for each block, the working variables (a, b, c, d, e, f, g, h) are added to the hash values from the previous block (h0, h1, h2, h3, h4, h5, h6, h7) to produce the new hash values.

4. Final Output Digest

After processing all blocks, the final hash is produced by concatenating the eight hash values (h0 to h7).

In this exercise, after understanding all the steps of the algorithm, you are required to study the provided `sha256_algorithm.c` file and understand how it implements the steps that we have just seen. It implements dynamic memory allocation to accept a string of any size and computes the SHA256 output hash of the string.

Then you can compile the file using `gcc` by navigating to the directory of the file and running,

```
gcc sha256_algorithm.c
./a.out
```

The program will take any input string as an input and provide the output SHA256 hash. You can play with it and read the source code to properly understand how the algorithm is implemented in code.

Now enable the debugging statements at each stage and visualize how the internal state changes with each step of the program.

Exercise 2: Interactive SHA256 Algorithm Visualization

After looking at a C program, we will look at a visual example using a wonderful online tool that you can access at <https://sha256algorithm.com>. We will be hashing the string “Hello World” and visually look at how these different steps take place in the SHA256 algorithm that we have previously studied and seen an implementation in C. You are required to access the calculator and visualize step by step how “Hello World” is transformed into a 256-bit hash. You will get the following final hash,

```
A591A6D40BF420404A011733CFB7B190D62C65BF0BCDA32B57B277D9AD9F146E
```

You need to go back to the C program and hash the same string with the C program from the previous exercise, and you will also verify that both the hashes match. The final hash of the C program can be seen below,

```
shahab@shahab-HP:~/OS/Linux/Project/sha256_algorithm$ gcc sha256_accelerator.c
shahab@shahab-HP:~/OS/Linux/Project/sha256_algorithm$ ./a.out
Enter string to hash: Hello World
SHA256 Digest: A591A6D40BF420404A011733CFB7B190D62C65BF0BCDA32B57B277D9AD9F146E
shahab@shahab-HP:~/OS/Linux/Project/sha256_algorithm$
```

At the end, you will have visualized and understood all the steps you read here, and also get an understanding of how the algorithm can be implemented in C.

Lab 2: Exporting a custom cryptocore to our QEMU RISC-V machine

In this lab, you will integrate a SHA256 cryptographic accelerator core into the QEMU RISC-V emulator environment that we had built in lab 0. This practical exercise is designed to enhance your understanding of how heterogeneous architectures are organized and how you can integrate a hardware device in an existing machine inside of QEMU. You will work on configuring QEMU to simulate the RISC-V architecture with an additional cryptographic core.

Inside QEMU, all devices are represented as objects using the QEMU Object Model (QOM). This model follows the object oriented programming (OOP) approach and provides the framework to register user created types where a device is an object inheriting from a parent object.

Exercise 1: Understanding the Cryptocore Model in QOM

In our case, from the QOM point of view our device `SHA256_DEVICE` is a child object inheriting from the system bus parent object of the QEMU RISC-V machine. Take a look at the provided `sha256_accelerator.c` file, specifically the `TypeInfo` structure which will be used to register our custom cryptocore.

```
static TypeInfo sha256_device_info = {
    .name = TYPE_SHA256_DEVICE,
    .parent = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(SHA256DeviceState),
    .instance_init = sha_instance_init,
};
```

There is also a device state structure that keeps track of the state of the device within the board (virtual machine), including device specific registers.

```
struct SHA256DeviceState {
    SysBusDevice parent_obj;
    MemoryRegion iomem;           // Memory region for device I/O
    char inputBuffer[inputBufferSize]; // Buffer to store input data
    uint8_t outputBuffer[outputBufferSize]; // Output hash buffer
    uint32_t control;             // Control register to manage the device
    uint32_t status;             // Status register to indicate device state
};
```

Finally, the cryptcore is initialized through the following instance initialization,

```
static void sha_instance_init(Object *obj) {
    SHA256DeviceState *s = SHA256_DEVICE(obj);

    /* allocate memory map region */
    memory_region_init_io(&s->iomem, obj, &sha_device_ops, s,
"sha256_device", 0x1000);
    sysbus_init_mmio(SYS_BUS_DEVICE(s), &s->iomem);

    // Initialize the state of the device
    s->status = 0;           // Set initial status as 0 (device idle)
    s->control = 0;          // Set control register as 0 initially
    memset(s->inputBuffer, 0, inputBufferSize); // Clear input buffer
    memset(s->outputBuffer, 0, outputBufferSize * sizeof(uint8_t));
}
```

Here we initialize our registers, allocate 0x1000 in memory using the `memory_region_init_io` function depending on the sizes of various registers, mainly the input register, and register the device on the system bus.

Exercise 2: Registering the cryptcore as a QEMU Device

As we know our target virtual board in QEMU, which is the RISC-V board defined as `qemu_riscv64_virt_defconfig`, we need to add our cryptcore to this board. QEMU uses the Kconfig and Meson build method to select and build dependencies for specific configurations, so we need to add our cryptcore to these dependencies. Navigate to the `qemu/hw/riscv/` directory and open the Kconfig file that we will edit. Inside that file, file the `config RISC_VIRT` entry, which should look like this,

```
config RISC_VIRT
    bool
    imply PCI_DEVICES
    imply TEST_DEVICES
    select GOLDFISH_RTC
    select MSI_NONBROKEN
    ...
    select SIFIVE_PLIC
    select SIFIVE_TEST
    select VIRTIO_MMIO
    select FW_CFG_DMA
```

At the end of this, add the following entry,

```
config RISC_VIRT
    bool
    ...
    select FW_CFG_DMA
    select SHA256_DEVICE
```

Now we also need to make the cryptcore visible to the Meson build system. For that, navigate to `qemu/hw/misc` and open the `meson.build` file. In the beginning you will see the following entries,

```
system_ss.add(when: 'CONFIG_APPLESMC', if_true: files('applesmc.c'))
system_ss.add(when: 'CONFIG_EDU', if_true: files('edu.c'))
system_ss.add(when: 'CONFIG_FW_CFG_DMA', if_true: files('vmcoreinfo.c'))
...
system_ss.add(when: 'CONFIG_EMPTY_SLOT', if_true: files('empty_slot.c'))
system_ss.add(when: 'CONFIG_LED', if_true: files('led.c'))
system_ss.add(when: 'CONFIG_PVPANIC_COMMON', if_true: files('pvpanic.c'))
```

In here, we need to add an entry for our cryptcore, and we can do that as follows,

```
system_ss.add(when: 'CONFIG_APPLESMC', if_true: files('applesmc.c'))
system_ss.add(when: 'CONFIG_SHA256_DEVICE', if_true:
files('sha256_accelerator.c'))
system_ss.add(when: 'CONFIG_EDU', if_true: files('edu.c'))
system_ss.add(when: 'CONFIG_FW_CFG_DMA', if_true: files('vmcoreinfo.c'))
...
system_ss.add(when: 'CONFIG_EMPTY_SLOT', if_true: files('empty_slot.c'))
system_ss.add(when: 'CONFIG_LED', if_true: files('led.c'))
system_ss.add(when: 'CONFIG_PVPANIC_COMMON', if_true: files('pvpanic.c'))
```

Now in the same directory we need to open the `Kconfig` file. Inside it you will see such entries,

```
config ARMSSE_CPUID
    bool

config ARMSSE_MHU
    bool

config ARMSSE_CPU_PWRCTRL
    bool

config MAX111X
    bool
```

You need to include the entry for our cryptcore as follows,

```
config ARMSSE_CPUID
    bool

config ARMSSE_MHU
    bool

config ARMSSE_CPU_PWRCTRL
    bool

config SHA256_DEVICE
    bool

config MAX111X
    bool
```

The final file to be edited, `virt.c` is in the directory `qemu/hw/riscv`. In here, first of all we will have to include the header file of our device,

```
#include "hw/misc/sha256_accelerator.h"
```

Next, as the crypto core is a memory mapped io, we have to assign space to it in the board memory map, which is defined below,


```
static const MemMapEntry virt_memmap[] = {
    [VIRT_DEBUG] = { 0x0, 0x100 },
    [VIRT_MROM] = { 0x1000, 0xf000 },
    [VIRT_TEST] = { 0x100000, 0x1000 },
    [VIRT_RTC] = { 0x101000, 0x1000 },
    [VIRT_CLINT] = { 0x2000000, 0x10000 },
    [VIRT_PCIE_PIO] = { 0x3000000, 0x10000 },
    ...
    [VIRT_PCIE_MMIO] = { 0x40000000, 0x40000000 },
    [VIRT_DRAM] = { 0x80000000, 0x0 },
};
```

Here, we will add our device to the memory map of the board to make space for it.

```
static const MemMapEntry virt_memmap[] = {
    [VIRT_DEBUG] = { 0x0, 0x100 },
    [VIRT_MROM] = { 0x1000, 0xf000 },
    [VIRT_TEST] = { 0x100000, 0x1000 },
    [VIRT_RTC] = { 0x101000, 0x1000 },
    [VIRT_CLINT] = { 0x2000000, 0x10000 },
    [VIRT_PCIE_PIO] = { 0x3000000, 0x10000 },
    [VIRT_SHA256_DEVICE] = { 0x5000000, 0x1000 },
    ...
    [VIRT_PCIE_MMIO] = { 0x40000000, 0x40000000 },
    [VIRT_DRAM] = { 0x80000000, 0x0 },
};
```

Afterwards, we now need to instantiate the device when the board is initialized. This includes using the memory map address and size we defined earlier, associating it with the system bus parent object, and adding it to the board device tree while setting up the device tree properties. We can add this after any such device creation function and before the `virt_machine_init` function, for example we can add it after,

```
static FWCfgState *create_fw_cfg(const MachineState *mc)
```

We add the following function,

```

static void create_sha256_device(const MachineState *mc)
{
    hwaddr base = virt_memmap[VIRT_SHA256_DEVICE].base;
    hwaddr size = virt_memmap[VIRT_SHA256_DEVICE].size;
    char *nodename;

    // Create the device, associating it with the base address and
interrupt
    sysbus_create_simple("sha256_device", base, NULL);

    // Construct the node name for the device tree
    nodename = g_strdup_printf("/sha256-device-nodename@%" PRIx64, base);

    // Add the device node to the device tree
    qemu_fdt_add_subnode(mc->fdt, nodename);

    // Set properties in the device tree
    qemu_fdt_setprop_string(mc->fdt, nodename, "compatible",
"sha256_accelerator");
    qemu_fdt_setprop_sized_cells(mc->fdt, nodename, "reg", 2, base, 2,
size);
    g_free(nodename);
}

```

Finally, inside the `virt_machine_init` function we have to call this function that we have created, passing the machine state pointer variable. In this function we have to do this before `ricscv_load_fdt` is called, otherwise the device tree can't be altered and our cryptcore will not be added. We will call our function just after the following piece of code,

```

s->fw_cfg = create_fw_cfg(machine);
rom_set_fw(s->fw_cfg);

```

To do this we will add the following line,

```

create_sha256_device(machine);

```

We also need to add the `VIRT_SHA256_DEVICE` enum to the `virt.h` header file. To do this, inside the `qemu` directory, move to the directory `/include/hw/riscv/` and inside the `virt.h` file locate the following enum,

```
enum {  
    VIRT_DEBUG,  
    VIRT_MROM,  
    VIRT_TEST,  
    ...  
    VIRT_PCIE_MMIO,  
    VIRT_PCIE_PIO,  
    VIRT_PCIE_ECAM  
};
```

Add the following to the end of the enum as follows,

```
enum {  
    VIRT_DEBUG,  
    VIRT_MROM,  
    VIRT_TEST,  
    ...  
    VIRT_PCIE_MMIO,  
    VIRT_PCIE_PIO,  
    VIRT_PCIE_ECAM,  
    VIRT_SHA256_DEVICE  
};
```

This process finishes all the changes to be made to the virtual board to register the cryptcore as a device. In the end, we just have to add the `sha256_accelerator.c` and `sha256_accelerator.h` files to the `qemu/hw/misc` directory. Now in the `qemu` directory, you have to run the following commands,

```
make clean  
make
```

This will build our RISC-V QEMU machine again, this time with the SHA256 cryptcore as a part of the hardware architecture, so we can interact with it from the userspace with the help of a Linux device driver. After the `make` process is complete, we can boot inside of our RISC-V machine through the Buildroot script we created in Lab 0. The BusyBox init program of Buildroot has a useful way of interacting directly with hardware, using which we can test if we have successfully added our cryptcore.

```
busybox devmem 0x5000000
```

The command directly accesses the provided memory location, which is the base address of our cryptocore which we defined in the board's memory map, along with the offset of the ID register of the crypto core which is 0x0. This should provide you with the hardcoded ID of the cryptocore, which confirms that we have successfully registered the cryptocore.

```
Welcome to Buildroot
buildroot login: root
# busybox devmem 0x5000000
0xFEEEDCAFE
#
```

We can also see our device if we run the following commands,

```
cd ../proc/device-tree/
ls
```

Here you can also see the SHA256 cryptocore added to the device tree of the virtual board.

```
Welcome to Buildroot
buildroot login: root
# cd ../proc/device-tree/
# ls
#address-cells          memory@80000000
#size-cells             model
chosen                 name
compatible             reserved-memory
cpus                   sha256-device-nodename@5000000
fw-cfg@10100000        soc
#
```

Lab 3: Device Driver Development and Configuration

In this lab you will look at how the device driver for the cryptocore can be developed to interact with the hardware through the userspace. We will look at a basic Linux Kernel Module, how it is written, compiled and loaded into the Linux kernel. We will then proceed with a driver for the SHA256 cryptocore and look at that.

Exercise 1: Developing and Loading a Simple Linux Kernel Module

To get started with developing and interacting with Linux kernel modules, you firstly need a package that allows you to do so. Run the following command,

```
sudo apt-get install build-essential kmod
```

This will give you access to different kernel module commands that you will need. For instance, the following commands give you a list of the modules that are currently loaded in your kernel.

```
sudo lsmod
```

This gives you the module name, size and how many devices are using it. Now before moving on to develop your first LKM, you also need to install header files for the kernel. To do that, you need to run,

```
sudo apt-get update  
apt-cache search linux-headers-`uname -r`
```

Now you need to get the kernel version, which can be done using,

```
uname -r
```

Now whatever your kernel version is, you need to run the following command writing your correct kernel version,

```
sudo apt-get install kmod linux-headers-yourkernelversion
```

```
shahab@shahab-HP:~$ uname -r
6.2.0-36-generic
shahab@shahab-HP:~$ sudo apt-get install kmod linux-headers-6.2.0-36-generic
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
kmod is already the newest version (29-1ubuntu1).
linux-headers-6.2.0-36-generic is already the newest version (6.2.0-36.37~22.04.1).
0 upgraded, 0 newly installed, 0 to remove and 3 not upgraded.
shahab@shahab-HP:~$
```

This will successfully install kernel headers. Now we can move on to writing our LKM, which is the simplest Hello Kernel LKM. Take a look at the dummy.c file provided for this exercise, it includes the kernel headers,

```
#include <linux/module.h>
#include <linux/init.h>
```

The metadata for the LKM,

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("SHAHABUDDIN");
MODULE_DESCRIPTION("Dummy LKM");
```

Initializing and exit functions,

```
static int __init ModuleInit(void) {
    printk("Hello, kernel!\n");
    return 0;
}

static void __exit ModuleExit(void) {
    printk("Goodbye, kernel!\n");
}
```

And the passing of those functions to the kernel functions,

```
module_init(ModuleInit);
module_exit(ModuleExit);
```

This simple dummy LKM can now be compiled and loaded into the kernel. For this we will need a Makefile, which is also included. It should be noted that the indentations in the Makefile are tabs and not spaces.

```
obj-m+=dummy.o

PWD:=$(CURDIR)

all:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build/ M=$(PWD) clean
```

Running `make` in this directory will compile the LKM for your kernel version and similarly running `make clean` will remove all of the generated files. Anyways, after running `make` you should have a `dummy.ko` file. We can find information about this module using this command,

```
modinfo dummy.ko
```

Now we can insert our module into the kernel. However, before that you need to understand that modern computers come configured with UEFI SecureBoot, which only allows trusted software distributed by OEMs. Linux distributions like Ubuntu ship with the kernel configured to support SecureBoot, but then in this case kernel modules require a signed security key to be able to load into the kernel. At this point attempting to insert this LKM will result in: “ERROR: could not insert module”. The simplest workaround for this problem is disabling UEFI SecureBoot from the boot menu of your PC. The alternative of generating keys, system key installation, and module signing can be done but is out of the scope of this exercise. Once you disable SecureBoot, you can run,

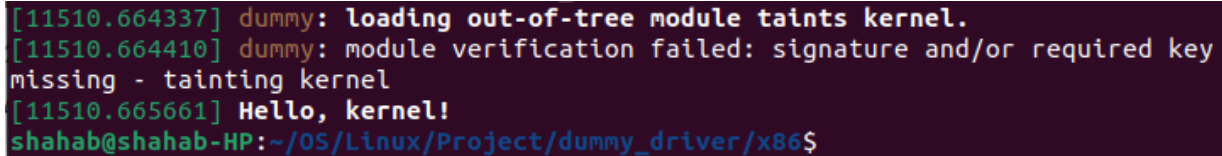
```
sudo insmod dummy.ko
```

This will not display any message, but now we can find our module using `lsmod` again. A more appropriate command to filter out and see only our module can be run now,

```
sudo lsmod | grep dummy
```

We can also check the kernel logs,

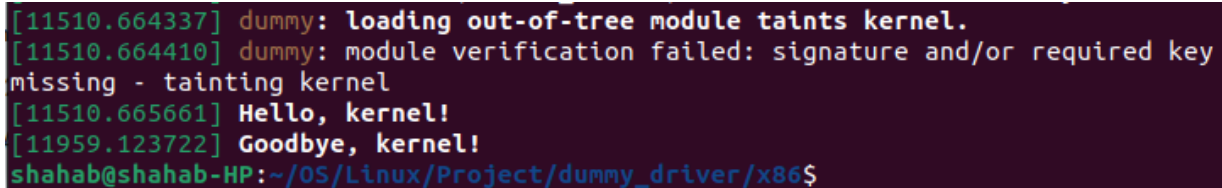
```
sudo dmesg
```

A terminal window with a dark purple background and light green text. It shows the output of the 'dmesg' command. The logs include: '[11510.664337] dummy: loading out-of-tree module taints kernel.', '[11510.664410] dummy: module verification failed: signature and/or required key missing - tainting kernel', and '[11510.665661] Hello, kernel!'. The prompt at the bottom is 'shahab@shahab-HP:~/OS/Linux/Project/dummy_driver/x86\$'.

Here we can see our printed message in the kernel logs, showing our LKM is successfully loaded. We also see two messages regarding tainting the kernel, however we can ignore them as they appear when an out of tree module is inserted into the kernel. For now, we can now look at removing the module from the kernel. For this we can run,

```
sudo rmmod dummy
```

Now if we view the kernel logs again using the previous `dmesg` command, we can see the module exit message printed which means the module has been successfully removed.

A terminal window with a dark purple background and light green text. It shows the output of the 'dmesg' command. The logs include: '[11510.664337] dummy: loading out-of-tree module taints kernel.', '[11510.664410] dummy: module verification failed: signature and/or required key missing - tainting kernel', '[11510.665661] Hello, kernel!', and '[11959.123722] Goodbye, kernel!'. The prompt at the bottom is 'shahab@shahab-HP:~/OS/Linux/Project/dummy_driver/x86\$'.

We can also run `lsmod` with `grep` again to find the dummy module but we will see nothing, confirming it has been removed from the kernel.

Exercise 2: Compiling and loading the Cryptocore Device Driver

Now we can move on to look at the device driver for our SHA256 cryptocore, which is also just another Linux kernel module, so it follows the same design principles as our dummy LKM, but it also includes some extra key features that are necessary for interacting with our virtual hardware core.

In Linux physical devices are represented as files, and so if a device driver has to read an input to the device it reads from the device file and to write an output it writes to the device file. This interaction is also streamlined for sending commands to a device with a

special function called `ioctl` (Input Output ConTrol), and every device can have its own `ioctl` commands that are configured through the device driver's `ioctl` function.

- The device driver for the cryptocore includes read and write function to read from and write to the input and output buffers of the core.
- The driver also has `ioctl` commands which the userspace application can use to send commands to the device.
- A driver also has a major number, which gives it a unique identifier within the kernel. Therefore, static allocation of major numbers is problematic as multiple kernel modules might be assigned the same major. This can be fixed with dynamic allocation, which is also included in the cryptocore device driver.

You are required to study and try to understand the provided cryptocore driver. It can be compiled and loaded to the kernel using the same method as we did in the previous exercise. One thing to note is that in the **sha_driver.c** file, line 350 needs to be commented and line 353 uncommented when compiling for x86 and not buildroot due to the difference in kernel version.

```
shahab@shahab-HP:~/OS/Linux/Project/sha_driver/v1/x86$ sudo insmod sha_driver.ko
shahab@shahab-HP:~/OS/Linux/Project/sha_driver/v1/x86$ sudo lsmod | grep sha_driver

sha_driver                16384  0
```

The kernel logs can also be viewed to see the driver is loaded successfully.

```
[14857.031201] SHA256: Initializing the driver
[14857.031296] SHA256 driver loaded with major 510
shahab@shahab-HP:~/OS/Linux/Project/sha_driver/v1/x86$
```

Upon removing the driver, we can see that it exists correctly,

```
[14857.031201] SHA256: Initializing the driver
[14857.031296] SHA256 driver loaded with major 510
[15010.615813] audit: type=1107 audit(1721588840.181:85): pid=631 uid=102 auid=42949
67295 ses=4294967295 subj=unconfined msg='apparmor="DENIED" operation="dbus_method_c
all" bus="system" path="/org/freedesktop/timedate1" interface="org.freedesktop.DBus
.Properties" member="GetAll" mask="send" name=":1.212" pid=2832 label="snap.firefox.
firefox" peer_pid=18740 peer_label="unconfined"
exe="/usr/bin/dbus-daemon" sauid=102 hostname=? addr=? terminal=?'
[15053.760956] rtw89_8852ae 0000:01:00.0: AMD-Vi: Event logged [IO_PAGE_FAULT domain
=0x000b address=0x0 flags=0x0000]
[15120.381275] SHA256: Exiting the driver
[15120.381359] SHA256: driver unregistered
shahab@shahab-HP:~/OS/Linux/Project/sha_driver/v1/x86$
```

Lab 4: Cross-compilation for RISC-V

Till now we have only compiled and tested our programs on the host machine which is probably running on x86 or ARM architecture. However, since the goal is to test the virtual hardware for RISC-V ISA, we have to learn how to compile everything so that it runs on RISC-V. Cross-compilation is essential when developing software for architectures that are different from the host machine, such as RISC-V. This process allows you to build applications and drivers on a more powerful machine that can then be executed on the target architecture.

Exercise 1: Cross-compilation toolchains and Buildroot

Cross-compiler

Cross-compilation for RISC-V involves using a RISC-V cross-compiler toolchain and library. A cross-compiler is basically a compiler that generates a binary for a target architecture which is different from the host architecture that the compiler is running on, like for example compiling for RISC-V on x86 requires a cross-compiler. Therefore, we will need something like the RISC-V GNU Compiler Toolchain, which can then be used to cross-compile binaries for RISC-V on our PC.

However, Buildroot simplifies this process as it already includes a RISC-V toolchain, as it needs to cross-compile the Linux kernel for our RISC-V virtual machine. We can utilize this compiler to cross-compile any program we want for RISC-V that will run perfectly well on our virtual QEMU machine. Inside the buildroot directory, if you navigate to `output/host/` you will find **riscv64-buildroot-linux-gnu**. This is the cross-compiler used by buildroot for our specific configuration.

```
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output$ tree host -L 1
host
├── bin
├── etc
├── include
├── lib
├── lib64 -> lib
├── libexec
├── riscv64-buildroot-linux-gnu
├── sbin
├── share
└── usr -> .

10 directories, 0 files
```

And the binary for the cross-compiler can be found at `output/host/bin`, which in our case is **riscv64-buildroot-linux-gnu-gcc**. We can view the version details for this if we are present inside the directory, like we can do for gcc on the host PC. To do this, inside the bin directory, run

```
./riscv64-buildroot-linux-gnu-gcc --version
```

And you can see the details for the cross-compiler,

```
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output/host/bin$ ./riscv64-buildroot-linux-gnu-gcc --version
riscv64-buildroot-linux-gnu-gcc.br_real (Buildroot 2024.02-827-g7aa4c0d9cc) 12.3.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output/host/bin$
```

Exercise 2: Setting up the RISC-V Cross-compiler

Now that we have seen the cross-compiler, let's move on to cross-compiling a simple Hello World program. We first need to add the cross-compiler to the environment path variable, for convenience but also for later functionality. We can add to the path temporarily, which will only remain for that session, but for this exercise we will add it permanently. There are various ways to do this, one of the simplest is to open a new terminal and run the following command,

```
sudo nano /etc/environment
```

This will open the environment file which is located in the /etc directory, and here you will customize the PATH to add the cross-compiler using the absolute path for it. The file should look something like this,

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin"
```

At the end you will append the absolute path of the cross-compiler, for example in my case the absolute path for the cross compiler is,

```
/home/shahab/OS/Linux/Project/QEMU/buildroot/output/host/bin
```

Therefore, my PATH variable will look like this,

```
PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/home/shahab/OS/Linux/Project/QEMU/buildroot/output/host/bin"
```

Once done, you can save and exit. Now, when you check your PATH variable in a terminal,

```
echo $PATH
```

You will see the directory of the cross-compiler added to your PATH variable. This means the system can now access it from anywhere. Now if you check the version of the cross-compiler from any terminal or directory it will work,

```
riscv64-buildroot-linux-gnu-gcc --version
```

```
shahab@shahab-HP:~$ riscv64-buildroot-linux-gnu-gcc --version
riscv64-buildroot-linux-gnu-gcc.br_real (Buildroot 2024.02-827-g7aa4c0d9cc) 12.3.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

shahab@shahab-HP:~$
```

Exercise 3: Cross-compilation of a simple program and running in Buildroot

Now we can finally cross-compile our program. Let's take the following program,

```
#include <stdio.h>

int main() {
    printf("Hello Buildroot Linux\n");
    return 0;
}
```

First compile it for the host machine by running,

```
gcc hello.c -o hello_host
```

Now to run it we can simply invoke it and see the output.

```
shahab@shahab-HP:~/OS/Linux/Project/Hello World$ gcc hello.c -o hello_host
shahab@shahab-HP:~/OS/Linux/Project/Hello World$ ./hello_host
Hello Buildroot Linux
shahab@shahab-HP:~/OS/Linux/Project/Hello World$
```

Now however if we want to compile for the target (RISC-V) architecture, we will invoke the cross-compiler. You can do this by running,

```
riscv64-buildroot-linux-gnu-gcc hello.c -o hello_target
```

This will create a new executable, but if you try to run it, you will get an error. Try to run this new binary.

```
shahab@shahab-HP:~/OS/Linux/Project/Hello World$ riscv64-buildroot-linux-gnu-gcc
hello.c -o hello_target
shahab@shahab-HP:~/OS/Linux/Project/Hello World$ ls
hello.c  hello_host  hello_target
shahab@shahab-HP:~/OS/Linux/Project/Hello World$ ./hello_target
bash: ./hello_target: cannot execute binary file: Exec format error
shahab@shahab-HP:~/OS/Linux/Project/Hello World$
```

This is because you have cross-compiled generated an executable binary for RISC-V and it is not recognized by the x86 host machine. We can also verify this information by running the file command for both executables.

```
file hello_host
file hello_target
```

```
shahab@shahab-HP:~/OS/Linux/Project/Hello World$ file hello_host
hello_host: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=aa97094aa424e018
765ed1e73a16937ab2049759, for GNU/Linux 3.2.0, not stripped
shahab@shahab-HP:~/OS/Linux/Project/Hello World$ file hello_target
hello_target: ELF 64-bit LSB pie executable, UCB RISC-V, double-float ABI, versi
on 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-riscv64-lp64d.so.1, f
or GNU/Linux 6.6.0, not stripped
```

Observe how they are both 64-bit ELF files, but both for different architectures and compiled for a different compiler. Now we will move on to running this file on our virtual QEMU machine inside buildroot. To do this, we have to first mount the rootfs of the target machine on the host machine. To do this, navigate to the buildroot directory inside which go to `/output/images/` and create a directory `mnt`, then run the following command,

```
sudo mount -t ext2 -o rw,loop rootfs.ext2 mnt
```

This essentially mounts the root filesystem of the target machine on a directory and allows

access to read or write files to the rootfs of the target QEMU machine, opening us to a wide range of new functionality. Now you can navigate to the **mnt** directory, access the root directory and copy any files you need on the target machine. For this example we will copy the `hello_host` and `hello_target` executable binaries for this exercise. You can run,

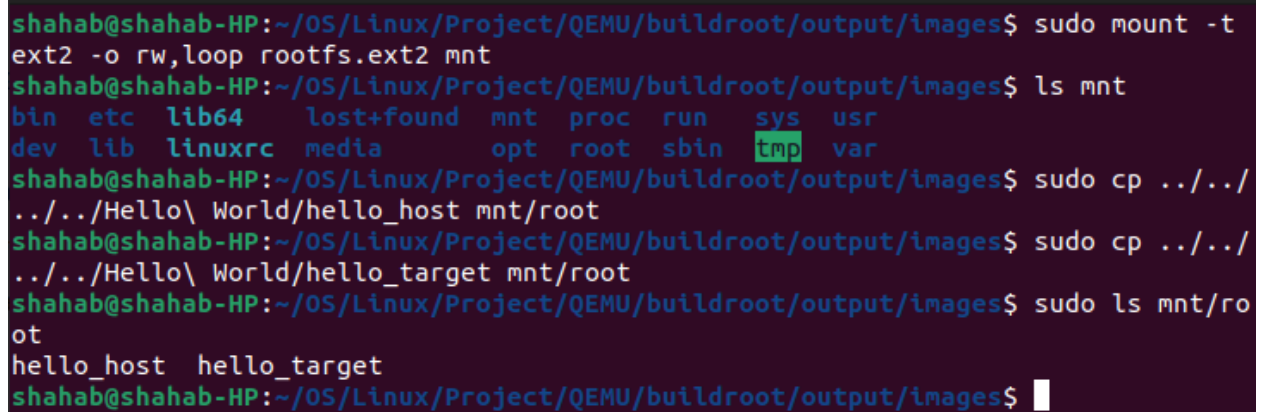
```
sudo cp <complete_path>/hello_host mnt/root
sudo cp <complete_path>/hello_target mnt/root
```

For example I ran the following according to my path of the binaries,

```
sudo cp ../../../../Hello\ World/hello_host mnt/root
sudo cp ../../../../Hello\ World/hello_target mnt/root
```

Then check it is successful by running,

```
sudo ls mnt/root
```



```
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output/images$ sudo mount -t
ext2 -o rw,loop rootfs.ext2 mnt
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output/images$ ls mnt
bin  etc  lib64  lost+found  mnt  proc  run  sys  usr
dev  lib  linuxrc  media      opt  root  sbin  tmp  var
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output/images$ sudo cp ../../
../../Hello\ World/hello_host mnt/root
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output/images$ sudo cp ../../
../../Hello\ World/hello_target mnt/root
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output/images$ sudo ls mnt/ro
ot
hello_host  hello_target
shahab@shahab-HP:~/OS/Linux/Project/QEMU/buildroot/output/images$
```

It can be seen that both files have been copied to the target machine. Now we can unmount the rootfs of the target machine. To do this you will run,

```
sudo umount mnt
```

We can finally run these executables on our QEMU virtual machine through Buildroot and see the result. Run the `start-qemu.sh` script and login. Here by running `ls` you can see the files you just added, and both executable binaries are present.

```
Welcome to Buildroot
buildroot login: root
# ls
hello_host    hello_target
#
```

Let's try to run them. First run the target binary to ensure it works correctly. And then you can run the host binary but it will not work.

```
Welcome to Buildroot
buildroot login: root
# ls
hello_host    hello_target
# ./hello_target
Hello Buildroot Linux
# ./hello_host
./hello_host: line 0: syntax error: unexpected word (expecting ")")
#
```

Exercise 4: Cryptocore device driver cross-compilation and insertion

In this exercise, we will follow the process seen in the last exercise, and take the cryptocore driver seen in lab 3 to cross-compile it for the target machine. To do this, we need to first make a few changes to the Makefile of the driver to enable cross-compilation.

```
MODULES := sha_driver.o

export ARCH := riscv
export CROSS_COMPILE := riscv64-buildroot-linux-gnu-
obj-m := $(MODULES)
KDIR :=
/home/shahab/OS/Linux/Project/QEMU/buildroot/output/build/linux-6.6.18

PWD:=$(CURDIR)

export
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    make -C $(KDIR) M=$(PWD) clean
```


The changes in this new Makefile include specifying the target architecture and cross-compiler toolchain prefix to be used. The KDIR variable is also changed to specify the location of the kernel for which we are compiling. Using this Makefile, we can run `make` and create the cross-compiled version of the driver. Follow the same procedure from the previous exercise to copy the cross-compiled driver to the root filesystem of the QEMU virtual machine. Now boot into your virtual machine again using the script.

```
Welcome to Buildroot
buildroot login: root
# ls
hello_host      hello_target    sha256_driver
# cd sha256_driver/riscv/
# ls
Makefile        sha_driver.c      sha_driver.mod.c
Module.symvers  sha_driver.ko     sha_driver.mod.o
modules.order   sha_driver.mod    sha_driver.o
#
```

Here you can see that the cross-compiled driver has been copied to the target machine. Now let's try loading it into the kernel as we did in lab 3 on our host machine. To do this run,

```
insmod sha_driver.ko
```

```
Welcome to Buildroot
buildroot login: root
# ls
hello_host      hello_target    sha256_driver
# cd sha256_driver/riscv/
# ls
Makefile        sha_driver.c      sha_driver.mod.c
Module.symvers  sha_driver.ko     sha_driver.mod.o
modules.order   sha_driver.mod    sha_driver.o
# insmod sha_driver.ko
[ 120.074003] sha_driver: loading out-of-tree module taints kernel.
[ 120.077495] SHA256: Initializing the driver
[ 120.078198] SHA256: Probe function called.
[ 120.080050] SHA256 Device created successfully
[ 120.080279] sha256_foo 5000000.sha256-device-nodename: SHA256 dev
zed
[ 120.080978] SHA256 driver loaded with major 243
#
```


As you can see, inside Buildroot we can see kernel logs as they come, which are indicated by the timestamp at the start. Here we see the same kernel taint message, which we can ignore. But we can also see some new messages, particularly the calling of the probe function and successful device creation. This is because the kernel module has detected a device with the same device tree properties as programmed into the device driver.

Similarly to view the inserted module in the module list, run

```
lsmod | grep sha_driver
```

Finally you can also confirm the device is initialized correctly by running,

```
cat /proc/devices
```



```
# lsmod | grep sha_driver
sha_driver          12288  0
# cat /proc/devices
Character devices:
 1 mem
 2 pty
 3 ttyp
 4 /dev/vc/0
 4 tty
 4 ttyS
 5 /dev/tty
 5 /dev/console
 5 /dev/ptmx
 7 vcs
10 misc
13 input
29 fb
128 ptm
136 pts
180 usb
189 usb_device
243 sha256
244 rpmb
245 ttySIF
```

Here you can see various devices along with their major numbers running on our virtual QEMU machine, and you can see the **sha256** device with major number 243 which was dynamically assigned when the device driver was loaded into the kernel. This successfully completes the cross-compilation and insertion of the device driver into the target kernel.

Lab 5: Testing SHA256 Cryptocore and Device Driver through Userspace

In this lab, you will conduct the final testing of the cryptocore and its associated device driver on the target RISC-V machine through userspace applications, which will be cross-compiled and integrated into the QEMU virtual machine. This testing is critical for functional verification of the device driver functions and the cryptocore behavior from the perspective of an end-user, providing an interface to the cryptographic hardware capabilities.

Exercise 1: Userspace Program to Read Cryptocore ID

Take a look at the provided userspace program `id_read.c`, which is a very basic program utilizing the device driver to read and print the ID register of the SHA256 device.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <stdint.h>

#define SHA256_IOC_MAGIC 'k'
#define SHA256_IOC_GET_ID _IOR(SHA256_IOC_MAGIC, 0, int)

int main() {
    int fd = open("/dev/sha2560", O_RDWR); // Open the device
    uint32_t id;

    if (fd < 0) {
        perror("Failed to open device");
        return -1;
    }

    if (ioctl(fd, SHA256_IOC_GET_ID, &id) == -1) {
        perror("Failed to get device ID");
        close(fd);
        return -1;
    }
    printf("Device ID: %x\n", id);
    close(fd);
    return 0;
}
```

This application opens the device file of the cryptocore, executing the `ioctl` command to fetch the ID information from the device. As in the previous lab, this program also needs to be cross-compiled and the executable binary needs to be copied to the target machine. Afterwards, you will load the device driver into the kernel as did in the last lab, and finally run the `id_read` executable.

```
Welcome to Buildroot
buildroot login: root
# ls
hello_host    hello_target  sha256_driver  userspace
# insmod sha256_driver/riscv/sha_driver.ko
[ 105.352036] SHA256: Initializing the driver
[ 105.352508] SHA256: Probe function called.
[ 105.353216] SHA256 Device created successfully
[ 105.353404] sha256_foo 50000000.sha256-device-nodename: SHA256 dev
zed
[ 105.353921] SHA256 driver loaded with major 243
# cd userspace/id_read/
# ls
id_read      id_read.c
# ./id_read
[ 123.624878] SHA256: Device file opened.
Device ID: feedcafe
[ 123.630395] SHA256: Device file closed.
#
```

The correct ID value of the cryptocore as seen in lab 2 is printed here, but this time through a userspace application, which uses the device driver that interfaces with the emulated hardware core through the Linux kernel. However, we can now create a userspace application to utilize the cryptographic capability of the hardware through the userspace.

Exercise 2: Userspace Program for SHA256 Hash Computation

In this exercise, we will test the complete cryptocore functionality by utilizing its hashing capability and compute the SHA256 digest for any given input string, and verify the correctness of the output hash. To do this, another userspace program `hash.c` is written which is provided for this exercise. It first reads and prints the ID code of the core, then takes an input string from the user and prints the computed SHA256 hash returned by the cryptocore.

To run the program, the same process of cross-compilation and copying has to be repeated as done previously. After running the program, you will get the ID code of the cryptcore and be prompted for an input string. Once you enter the input string, the application will print the final SHA256 digest received by the driver which reads it from the core device file and then the application exits. This can be run on any input string up to 1KB.

```
buildroot login: root
# ls
hello_host    hello_target  sha256_driver userspace
# insmod sha256_driver/riscv/sha_driver.ko
[ 21.099886] sha_driver: loading out-of-tree module taints kernel.
[ 21.107722] SHA256: Initializing the driver
[ 21.108520] SHA256: Probe function called.
[ 21.111366] SHA256 Device created successfully
[ 21.111742] sha256_foo 50000000.sha256-device-nodename: SHA256 device initiali
zed
[ 21.113094] SHA256 driver loaded with major 243
# cd userspace/id_read/
# cd ../hash/
# ls
hash    hash.c
# ./hash
[ 58.344330] SHA256: Device file opened.
Device ID: feedcafe
Enter a string to hash: Hello World
[ 61.363045] SHA256: Hashing process started.
The Final SHA256 Hash: a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9a
d9f146e
[ 61.364270] SHA256: Device file closed.
#
```

Conclusion

This concludes this comprehensive lab experience which was designed to teach development and testing of cryptographic accelerator cores and their associated device drivers within a simulated RISC-V environment. Throughout these labs, you have progressed from setting up your development environment and understanding the fundamentals of the SHA256 algorithm, to integrating a cryptographic core into QEMU, and finally to developing and testing the corresponding device driver through user-space applications.