LupIO: a collection of education-friendly I/O devices

Joël Porquet-Lupine
jporquet@ucdavis.edu
Department of Computer Science
University of California, Davis
Davis, California, USA

ABSTRACT

LupIO is a comprehensive and open-source collection of education-friendly I/O devices. This collection defines the interfaces of the most common devices found in modern RISC-based computers, and makes it possible to build complete systems using only LupIO devices, even complex multicore systems.

Each device interface is designed to be simple and clear, with an optimal balance between features and complexity. The register maps exposed by the devices are neatly organized by type and arranged consistently across devices. Developing implementations of LupIO devices, as well as corresponding device drivers, is meant to be straightforward and accessible to students at the undergraduate and graduate level.

As a proof of concept, LupIO was entirely implemented as virtual devices in QEMU, along with corresponding device drivers in Linux, and we were able to successfully boot a RISC-V based dual-core virtual machine.

The specifications are available at https://gitlab.com/luplab/lupio.

CCS CONCEPTS

• **Computer systems organization** → **Architectures**; • **Applied computing** → **Education**.

ACM Reference Format:
Joël Porquet-Lupine. 2021. LupIO: a collection of education-friendly I/O devices. In *Proceedings of Workshop on Computer Architecture Education (WCAE'21).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

There has always been a difficult tension in teaching between showing real-life products or simplified versions of them. This has been particularly true in computer architecture, especially at undergraduate level.

Take, for example, the choice of a processor architecture in the students' first computer organization course [5]. On one hand, if instructors teach, say, the Intel x86 assembly language, then they guarantee that students will be learning an immediately transferable skill, as most commodity computers are based on x86 processors. However, teaching x86 assembly, with its 40+ years of legacy, is also guaranteed to be painful—unless simplified [4]—for both the instructors and the students. On the other hand, if instructors instead choose a fictitious, much simpler processor such as CUSP [9], then the learning experience will certainly be a lot easier. However, it may be more difficult for students to later adapt their newly acquired knowledge to real processors, such as x86 processors. It is therefore no surprise that many RISC processors have been successful teaching artifacts [6, 11], because they are education-friendly while concurrently evolving into actual industrial products [15].

While the tension regarding the choice of a processor architecture has been well explored, the choice of I/O devices has not yet received the same kind of attention. The dilemma is very similar though, since real-life I/O devices are also often too complicated for educational purposes. Consider the venerable line of 8250/16550-compatible universal asynchronous receivers-transmitters (UART) [13] for simple serial communication, probably some of the most ubiquitous devices found across computer systems. A UART contains 12 internal registers but only 8 available memory addresses, so students would need to learn that some registers are multiplexed behind the same memory addresses using a switch bit. UARTs also provide many complex features such as the ability to configure the communication parameters (e.g., parity, speed, number of data bits being transferred, error control, etc.), FIFO buffers, direct memory access (DMA) capabilities, etc. Although it could be possible to help beginner students work with a UART by providing them with a detailed programming sequence, it is unlikely that they would truly understand how the device operates and how to properly use it.

This paper introduces **LupIO**, a comprehensive and open-source collection of education-friendly I/O devices. This collection defines the interfaces of the most common devices found in modern RISC-based computers, and makes it possible to build complete systems using only LupIO devices, even complex symmetric multiprocessor (SMP) systems. LupIO includes devices such as an interrupt controller, a timer, an interprocessor interrupt controller, a terminal, a generic block device, etc. LupIO devices are intended to be processor-agnostic, so they should be usable with any processor architecture supporting memory-mapped devices (e.g., RISC-V, ARM, MIPS, etc.). Each device interface is designed to be simple and clear, with an optimal balance between features and complexity. The register maps exposed by the devices are neatly organized by type (e.g., data, control, and status) and arranged consistently across devices, in order to ease their programmability. Developing implementations of LupIO devices, as well as corresponding device drivers, is meant to be straightforward.

The paper is organized as follows. First, in section 2, we provide an overview of existing fictitious devices, including their strengths and weaknesses. Then, in section 3, we explain the goals and design choices behind the LupIO collection. In section 4, we introduce the complete collection of LupIO devices. In section 5, we present a

WCAE'21, June 17, 2021, Virtual Event, Valencia, Spain

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

complete implementation of LupIO devices, both virtual devices and device drivers. In section 6, we suggest possible ways LupIO devices can be used throughout a typical CS curriculum, both at undergraduate and graduate levels. Finally, we conclude and discuss our next steps in section 7.

2 RELATED WORK

Fictitious, simpler I/O devices have existed for a long time, both for educational purposes and industrial applications. But so far no existing project has provided the specifications for a comprehensive collection of educational I/O devices able to fully equip a complete SMP system.

Educational simulators such as SPIM [12] or MARS [16], only define one or two memory-mapped I/O devices at most. While it is usually adequate to illustrate the interactions between software and hardware in typical computer organization courses at the undergraduate level, such simulators cannot be used later in the curriculum, for example in operating systems courses or more advanced computer architecture courses.

More advanced simulators, such as System/161 [10] which was typically developed for operating system courses at the undergraduate level, sometimes provide a more complete collection of virtual I/O devices. However, in the case of the MIPS-based System/161, the devices were only designed for this simulator and would not necessarily work seamlessly with other processor architectures.

Targeting industrial applications, the popular VirtIO project [7] defines a solid list of virtual devices, which includes a network device, a block device, a console device, etc. However, these devices are all strictly I/O oriented and do not include other devices necessary to computer systems, such as an interrupt controller, a timer, etc. Besides, the project's goal is focused entirely on simulation speed and has no educational aspirations. The implementations of these virtual devices or corresponding OS drivers definitely rivals real-life devices in terms of complexity, if not more.

Initially developed by Google for their Android emulator, Goldfish [8] probably includes the most complete collection of simple virtual devices, as it includes many typical devices (e.g., interrupt controller, terminal, timer, etc.). Although these devices were meant to offer simpler interfaces than realistic ones, the goal was not educational per se and it sometimes shows. For example, the interface across devices is not always consistent. Additionally, it is not certain one can build an entire SMP system using Goldfish devices only, and there is no generic block device—only a NAND flash device and a MultiMediaCard (MMC) device, both of which are probably too complex to teach at undergraduate level.

3 GOALS AND DESIGN CHOICES

The overarching goal of LupIO devices is to provide a collection of **education-friendly** I/O devices, which can be used from the first computer organization course at the undergraduate level up to advanced parallel programming courses at the graduate level. Balancing these accessibility and features requirements had deep impacts on how the device interfaces were designed.

3.1 Collection

The LupIO collection of devices is complete and generic enough to build a fully functional computer system only using LupIO devices and based around any education-friendly RISC processor (such as RISC-V or MIPS32). It includes core devices (such as an interrupt controller or a timer) as well as general I/O devices (such as a block device, a real-time clock, or a terminal). All of these devices are agnostic to the processor architecture used in the system.

As we acknowledge that multicore systems are becoming ubiquitous, parallel programming is also sure to become a pervasive topic, even at the undergraduate level. LupIO therefore supports SMP systems.

Real-life devices often offer multiple features which can be tricky to present to students. For example, interrupt controllers routinely embed timers and sometimes have support for inter-processor interrupts in SMP contexts. In a computer organization course, introducing such a device just to have students use a single timer would be too complicated. In LupIO, each device provides only one service.

3.2 Register map

When the concept of register maps is taught in class, students are usually presented with a neat classification of registers (e.g., data, control, status). But most actual device interfaces often combine control and status registers, which can be confusing. In LupIO, we define four distinct types of registers:

- Data registers are used to read data from or send data to the device. Input data registers are usually read-only, while output data registers are usually write-only.
- Control registers are used to configure the device, or initiate a specific I/O request. They are usually write-only.
- Status registers are used to provide status information about the device, or about a specific I/O request. They are usually read-only.
- Finally, optional configuration registers are used to give general characteristics about the device.

The set of internal registers is also laid out in a consistent manner across devices. In LupIO devices, the chosen order is: (1) configuration registers (if any), (2) data registers, (3) control registers, and (4) status registers. In addition to providing clarity, this ordering often has advantageous side-effects:

- The ordering often matches exactly how the device is to be used. Take the block device, for instance: (1) the device driver starts by reading the configuration register in order to discover the block device's properties; when the device driver wants to initiate a transfer request, (2) it first configures the data registers and then (3) starts the transfer by writing to the control register; finally, once the transfer is over, (4) the device driver reads the status register to acknowledge its completion status.
- Some devices offer partial functionality directly via the very first register of their register map. For example, the timer's first register provides a monotonic counter. This enables instructors to either gradually present how certain LupIO devices work, by revealing the register map step-by-step, or to simply use devices in their partial versions.

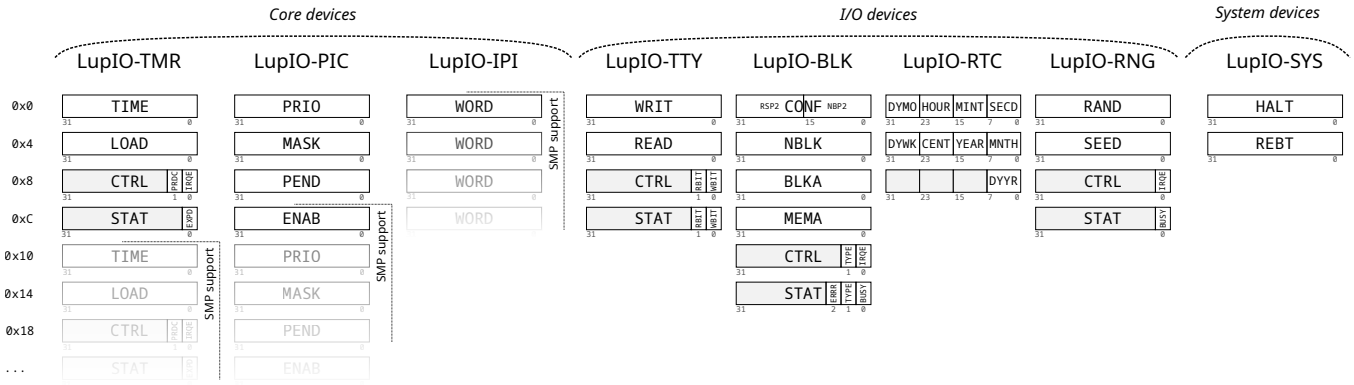

Figure 1: Register maps of all LupIO devices

The register map of each device does not present holes, even if the device is not used to the fullest of its capabilities. This design choice forced two important decisions. The first is that LupIO only supports a 32-bit physical address space (see subsection below for discussion about this particular point). The second is that LupIO devices that are compatible with SMP systems present a "multi-device" interface. This means that the regular mono-processor interface is simply duplicated as many times as the number of processor cores in the system. These devices typically have a 16-bytes long register map so the duplication is regular and holeless.

3.3 Address space

Although 64-bit computer systems are becoming increasingly pervasive, they are not education-friendly: working with 32-bit addresses on a blackboard remains significantly easier than with 64-bit addresses. Using a 64-bit address space would also be an issue for LupIO and the design choices presented so far.

Take the block device for instance, which has DMA capability. If it supported a 64-bit physical address space, then we would need two data registers for the memory address. The problem is that in a typical undergraduate-level computer organization course, instructors often still pick a 32-bit processor architecture for the sake of simplicity. In that case, it means that students would be instructed to ignore this second data register, which in our opinion is not education-friendly. Another possibility would be to move the data register for the 64-bit address extension, say at the end of the register map; this way students working with a 32-bit system would not need to see it. However, it would break our design choice of consistent register map organization. We therefore decided to only support a 32-bit address space.

Nonetheless, we do not believe this constraint is much of a limitation. If an instructor opts for a 64-bit processor to teach their class, it is very likely that the physical memory would be mapped in the first 4 GiB of the address space anyway. Our block device would, in that case, still remain compatible.

3.4 Implementation

LupIO devices are designed to be easily implemented as virtual devices, since instructors are more likely to use simulators or emulators than physical systems. To that effect, LupIO devices do not present any hardware-specific behaviors (e.g., speed, latency, bandwidth, etc.), which would otherwise be complicated to emulate in software.

Writing device drivers for LupIO is also meant to be a straightforward process. As LupIO is meant to be used in operating systems courses, students should be able to write some device drivers themselves.

Finally, it should be possible to implement LupIO devices in hardware. The interface of each device is therefore agnostic to the device's internal implementation, and allows typical behaviors found in physical devices, such as latency.

4 THE LUPIO COLLECTION OF DEVICES

As of today, the LupIO collection defines the specifications of eight devices. We present them in this section, loosely organized into four categories: core devices, I/O devices, system devices, and core devices for SMP support.

Figure 1 shows the complete register maps of all the LupIO devices. As per the design goals, the registers are organized by type, and in a consistent manner. Additionally, the names of registers and fields are exactly four characters in length and are also consistent across all devices.

4.1 Core devices

LupIO defines two core devices, which are indispensable to almost any mono-processor computer system: a timer and a programmable interrupt controller.

4.1.1 LupIO-TMR. This device provides a real-time counter, and a configurable timer offering periodic and one-shot modes. Its register map counts four 32-bit registers. The first register, `TIME`, is a read-only monotonic real-time counter. Its rate is implementation-dependent, and the value wraps around when it overflows. The second register, `LOAD`, is the reload value for the configurable timer. In one-shot mode, the timer is initialized with the reload value and then gets monotonically decremented until it expires when reaching zero. In periodic mode, the timer is automatically reloaded with the reload value upon expiration. The third register, `CTRL`, launches the timer when written to, unless the reload value is zero, in which case any currently running timer is immediately stopped.

If the IRQE bit is set, then the timer will raise an interrupt request (IRQ) whenever it expires. If the PRDC bit is set, then the timer is configured to be periodic instead of one-shot. Finally, the fourth register, STAT, returns the timer's status. If the EXPD bit is set, it means that a timer has expired and has not yet been acknowledged. Reading this register acknowledges the timer expiration and lowers the IRQ if it was raised.

4.1.2 LupIO-PIC. This device provides a programmable interrupt controller, which can manage input IRQs coming from up to 32 sources. In its non-SMP version, its register map counts three 32-bit registers. The first register, PRIO, returns the number of the highest-priority, unmasked, and active input IRQ. The highest-priority IRQ is from source #0, while the lowest-priority IRQ is from source #31. If no input IRQ is currently active, the register returns value 32. The second register, MASK, allows to mask or unmask each of the 32 sources individually. If bit #n of MASK is set, then IRQ #n is unmasked. Finally, the third register, PEND, returns the status of all active IRQs, regardless of their mask status. If bit #n of PEND is set, then IRQ #n is active.

4.2 I/O devices

LupIO defines four I/O devices that offer extra, often interactive, features to the computer system: a terminal, a block device, a real-time clock, and a random number generator.

4.2.1 LupIO-TTY. This device provides a terminal, which can transmit characters (e.g., to a screen) and receive characters (e.g., from a keyboard). This device has a single output IRQ, set as a logical OR between the transmitter and the receiver. Its register map counts four 32-bit registers. When writing to the first register, WRIT, the provided character is transmitted. When reading this register, it returns the last transmitted character and lowers the transmitter part of the device's IRQ if it was raised. If the last transmitted character has already been read, then the returned value is undefined. The second register, READ, returns the last received character, and lowers the receiver part of the device's IRQ if it was raised. If no character was received, then the returned value is undefined. The third register, CTRL, configures both transmitter and receiver. If the WBIT bit is set, then an IRQ will be raised once a character has been successfully transmitted. If the RBIT bit is set, then an IRQ will be raised when a character is received. Finally, the fourth register, STAT, returns the device's status. If the WBIT bit is set, then the device is ready to transmit a new character. If the RBIT bit is set, then the device has received a new character, which can be read via the READ register.

4.2.2 LupIO-BLK. This device provides a block device that can be used to add secondary storage to a computer system. Its register map counts six 32-bit registers. The first register, CONF, gives the configuration of the device via two 16-bit fields. The upper-half of the register, field BSP2, is the power of two representing the size of each block. For example, if this value is 9, then the block size is $2^9 = 512$ bytes. The lower-half of the register, field NBP2, is the power of two representing the number of blocks on the device. The next three registers, NBLK, BLKA, and MEMA, are used to configure a transfer between the block device and memory. NBLK is the amount of data to transfer (in blocks), BLKA is the address of the first block, and MEMA is the memory address of the buffer. Once a transfer is configured, the fifth register, CTRL, initiates the transfer when written to. If the IRQE bit is set, then an IRQ will be raised once the transfer is complete. If the TYPE bit is set, then the transfer is a write which means that NLBK blocks of data will be transferred from the memory, starting from address MEMA, to the block device, starting from block BLKA. If the TYPE bit is unset, then the transfer is a read, and the data will be transferred from the block device to the memory. Finally, the sixth register, STAT, returns the current status of the block device and of the last completed transfer. If the BUSY bit is set, then a transfer is currently ongoing and none of the other fields are relevant. The TYPE bit gives the type of the last completed transfer (write if set, read if unset). If the ERRR bit is set, then an error was encountered during the last transfer. Reading STAT lowers the IRQ if it was raised.

4.2.3 LupIO-RTC. This device provides a real-time clock, which is meant to supply the current date and time to the computer system in ISO 8601 format. Its register map counts nine 8-bit registers. The first three registers return the current time: SECD gives the second between 0 and 60 (where 60 is only used to denote an added leap second), MINT gives the minute between 0 and 59, and HOUR gives the hour between 0 and 23. The next four registers return the current date: DYMO gives the day of the month between 1 and 31, MNTH gives the month between 1 and 12, while the full year between 0 and 9999 is split between two registers, YEAR which gives the two lowest digits between 0 and 99 and CENT which gives the two highest digits between 0 and 99. Finally, the last two registers provide further information about the day: DYWK gives the day of the week between 1 and 7 (beginning with Monday and ending with Sunday), while DYYR gives the day of the year between 1 and 366 (366 for leap years).

4.2.4 LupIO-RNG. This device provides a basic random number generator. Its register map counts four 32-bit registers. The first register, RAND, returns a random number. If no random number is available, then the returned value is undefined. The second register, SEED, allows the user to configure the seed value for a sequence of random numbers. The third register, CTRL, configures the device. If the IRQE bit is set, then an IRQ is raised when a new random number is available. Finally, the fourth register, STAT, gives the device's status. If the BUSY bit is set, then no random number is currently available.

4.3 System devices

LupIO defines one system device, which provides the software with the ability to halt or reboot the computer system.

4.3.1 LupIO-SYS. This device provides a system controller to the executed software. Its register map counts two 32-bit registers. When writing to the first register, HALT, the computer system is halted. When writing to the second register, REBT, the computer system is rebooted. The interpretation of the value written in either register is left to the implementation. For example, if the computer system is emulated in software, then writing a value val to HALT could make the emulator call `exit(val)`. This behavior can be very convenient for automatic testing.

4.4 SMP support

LupIO extends two existing devices and provides one new device in order to support SMP systems.

4.4.1 LupIO-TMR. This device provides each processor core with its private timer by simply duplicating the timer register map presented above into as many instances as there are processor cores. Since the register map is 16 bytes long, the duplication is both easy to implement from the device's point of view, and easy to use from a device driver's point of view.

4.4.2 LupIO-PIC. This device is enhanced to accommodate two typical requirements related to IRQs in SMP systems. The first is that each processor core typically manages its own IRQs. To accomplish that, the device's register map presented above is duplicated into as many instances as there are processor cores. The second is that IRQs from the different sources can be routed to any processor core in the system. For that, the register map of each instance is increased with an additional (fourth) register, `ENAB`. Each bit of `ENAB` represents whether or not the corresponding IRQ source is routed to the processor core of that instance. If bit `#n` of `ENAB` is unset, then source `#n` will not be considered at all. Upon reset, the `ENAB` register belonging to the first register map instance is initialized to `0xFFFFFFFF`, which means that all IRQs are by default routed to processor core `#0`. Finally, since each register map instance is 16 bytes long, the same positive properties as discussed above for LupIO-TMR apply.

4.4.3 LupIO-IPI. This device provides a way for the kernel executing on a given processor core to send physical inter-processor interrupts (IPIs) to other processor cores, and force them to perform certain actions (such as stopping, rescheduling, or executing a certain kernel function). The register map only counts one register, `WORD`, and is duplicated into as many instances as there are processor cores. When writing a value into the `WORD` register of a certain instance, an IRQ is raised for the corresponding processor core. When reading the `WORD` register of a certain instance, its value is returned and the corresponding IRQ is lowered. If the register has already been read, then the returned value is undefined.

5 IMPLEMENTATION

In order to test the feasibility of the LupIO devices, we implemented them as virtual devices in machine emulator QEMU and wrote corresponding device drivers in the Linux kernel. As shown in figure 4, we were able to successfully boot a RISC-V based dual-core virtual machine.

5.1 Virtual devices implementation

QEMU [2] is a free and open source machine emulator. It can emulate plenty of processor architectures (ARM, x86, MIPS, RISC-V, etc.), and defines many emulation models of I/O devices.

We implemented all the LupIO devices in a self-contained and architecture-agnostic way in `qemu-src/hw/lupio`. Although implementing the internal logic behind each LupIO device was fairly straightforward, most of the difficulty was in figuring out how to interface them with QEMU's environment.

The implementation of memory-mapped virtual devices is twofold. First, one needs to initialize the device by resetting its internal registers (if any), defining its memory-mapped region, and attaching it to the memory bus. Second, one needs to define two functions, which determine the device's behavior when its register map is being accessed, on either a read or a write.

Listing 1 shows the read function's implementation for LupIO-RTC. The real-time is first retrieved by calling functions from QEMU's environment—returning the host computer's time—and then according to which register is being accessed, the proper value is returned in the format defined by LupIO-RTC's specifications.

One of LupIO's goals is that virtual devices should be easy to implement. While it is hard to formally prove the simplicity of a code implementation, one possible way is to count its number of lines of code (as measured by the tool sloccount [17]). Longer pieces of code usually tend to be more complex than shorter ones. Figure 2 shows, for each type of device, the number of lines of code for our LupIO virtual device as compared to other devices of the same type currently defined in QEMU. Our implementations consistently rank among the shortest[1].

Listing 1: Implementation of the read function in LupIO-RTC's QEMU virtual device

```
static uint64_t lupio_rtc_read(void *opaque, hwaddr addr, unsigned int size)
{
    uint32_t r = 0;
    uint64_t time_nsec;
    time_t time_sec;
    struct tm time_bd;

    /* Get real time in seconds */
    time_nsec = qemu_clock_get_ns(rtc_clock);
    time_sec = time_nsec / NANOSECONDS_PER_SECOND;
    /* Transform into broken-down time representation */
    gmtime_r(&time_sec, &time_bd);

    /* Determine which register was accessed */
    switch (addr) {
        case LUPIO_RTC_SECD:
            r = time_bd.tm_sec;                 /* 0-60 (for leap seconds) */
            break;
        case LUPIO_RTC_MINT:
            r = time_bd.tm_min;                 /* 0-59 */
            break;
        case LUPIO_RTC_HOUR:
            r = time_bd.tm_hour;                /* 0-23 */
            break;
        case LUPIO_RTC_DYMO:
            r = time_bd.tm_mday;                /* 1-31 */
            break;
        case LUPIO_RTC_MNTH:
            r = time_bd.tm_mon + 1;             /* 1-12 */
            break;
        case LUPIO_RTC_YEAR:
            r = (time_bd.tm_year + 1900) % 100; /* 0-99 */
            break;
        case LUPIO_RTC_CENT:
            r = (time_bd.tm_year + 1900) / 100; /* 0-99 */
            break;
        case LUPIO_RTC_DYWK:
            r = 1 + (time_bd.tm_wday + 6) % 7;  /* 1-7 (Monday is 1) */
            break;
        case LUPIO_RTC_DYYR:
            r = time_bd.tm_yday + 1;            /* 1-366 (for leap years) */
            break;
        default:
            qemu_log_mask(LOG_GUEST_ERROR, "%s:_Bad_offset_"TARGET_FMT_plx"\n",
                          __func__, addr);
            break;
    }
    return r;
}
```

[1]Note that: since IPI support is generally a part of interrupt controllers, we folded our LupIO-IPI virtual device into the same category; we were unable to compare LupIO-SYS as there are no other system controllers available in QEMU.

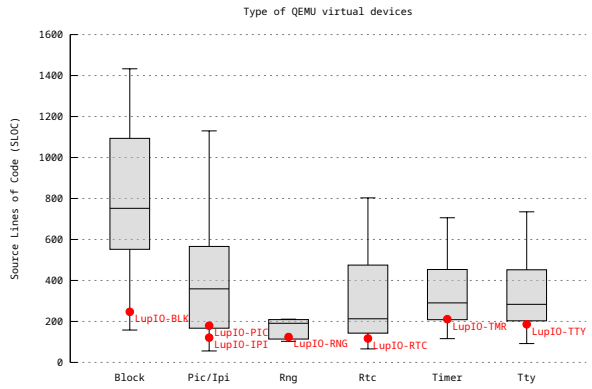

Figure 2: SLOC per type of QEMU virtual devices compare to LupIO-devices

5.2 Device drivers implementation

Similarly to the virtual devices implementation in QEMU, all the device drivers for Linux [14] were implemented in a self-contained and architecture-agnostic way in `linux-src/drivers/lupio`. The implementation of device drivers is usually two-fold. First, the device driver is initialized if the corresponding device is found in the platform's description (e.g., via its device tree). The device driver then maps the device in memory, and registers itself with the related Linux subsystem. Second, the device driver's callback functions are called when the subsystem wants to interact with the device.

Listing 2 shows the callback functions for LupIO-RTC's device driver. When retrieving the current time and date, the driver performs read memory accesses to the device.

Listing 2: Implementation of the read/write functions in LupIO-RTC's Linux device driver

```
static int lupio_rtc_read_time(struct device *dev, struct rtc_time *tm)
{
        struct lupio_rtc_data *rtc = dev_get_drvdata(dev);
        void __iomem *base = rtc->virt_base;

        /* Fill out `tm` object */
        tm->tm_sec      = readb(base + LUPIO_RTC_SECD);
        tm->tm_min      = readb(base + LUPIO_RTC_MINT);
        tm->tm_hour     = readb(base + LUPIO_RTC_HOUR);
        tm->tm_mday     = readb(base + LUPIO_RTC_DYMO);
        tm->tm_mon      = readb(base + LUPIO_RTC_MNTH) - 1;
        tm->tm_year     = readb(base + LUPIO_RTC_YEAR)
                + readb(base + LUPIO_RTC_CENT) * 100
                - 1900;
        tm->tm_wday     = readb(base + LUPIO_RTC_DYWK) % 6;
        tm->tm_yday     = readb(base + LUPIO_RTC_DYYR) - 1;
        tm->tm_isdst    = -1; /* Unavailable */

        return 0;
}

static int lupio_rtc_set_time(struct device *dev, struct rtc_time *tm)
{
        return -EOPNOTSUPP;
}

static const struct rtc_class_ops lupio_rtc_ops = {
        .read_time = lupio_rtc_read_time,
        .set_time = lupio_rtc_set_time,
};
```

Once again, although it would be difficult to prove that these device drivers are easy to implement, it is possible to measure the number of lines of code for each of them, and compare this result to similar Linux device drivers. Figure 3 shows, for each type of device, the number of lines of code for our LupIO device driver as compared to other device drivers of the same type currently defined in Linux. Our drivers consistently rank among the shortest[2].

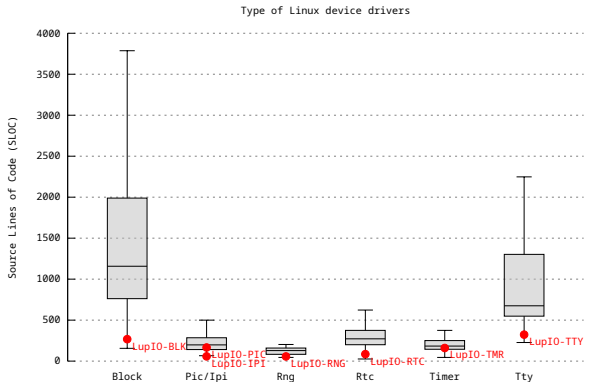

Figure 3: SLOC per type of Linux device drivers compare to LupIO-devices

5.3 RISC-V based platform

In order to build an entire virtual machine equipped with LupIO devices, we chose to use RISC-V [1] as the base processor. RISC-V supports memory-mapped devices by default, and conveniently exposes three input IRQs for each type of interrupt: inter-processor interrupts, timer interrupts, and general external interrupts. Given this configuration, it is possible to plug in our own core LupIO devices separately; respectively, LupIO-IPI, LupIO-TMR, and LupIO-PIC.

In QEMU, we defined a new RISC-V based platform called `lupv`, derived from the default RISC-V `virt` platform. The only hiccup was to expose the processor core's IRQ inputs in a generic way, so that our virtual devices could stay completely processor-agnostic. In Linux, the RISC-V port also required a few adaptations as the use of typical RISC-V devices (e.g., CLINT, PLIC) or bootloaders (e.g., OpenSBI) was hardcoded in many places. Once the code was made to be more generic, our LupIO device drivers could be plugged in seamlessly.

Figure 4 shows a partial but commented trace of our modified Linux booting on our LupIO-based QEMU virtual machine.

[2]Note that: since IPI support is generally a part of interrupt controllers, we folded our LupIO-IPI device driver into the same; there is no specific device driver for LupIO-SYS as it is supported by default on Linux with a generic driver.

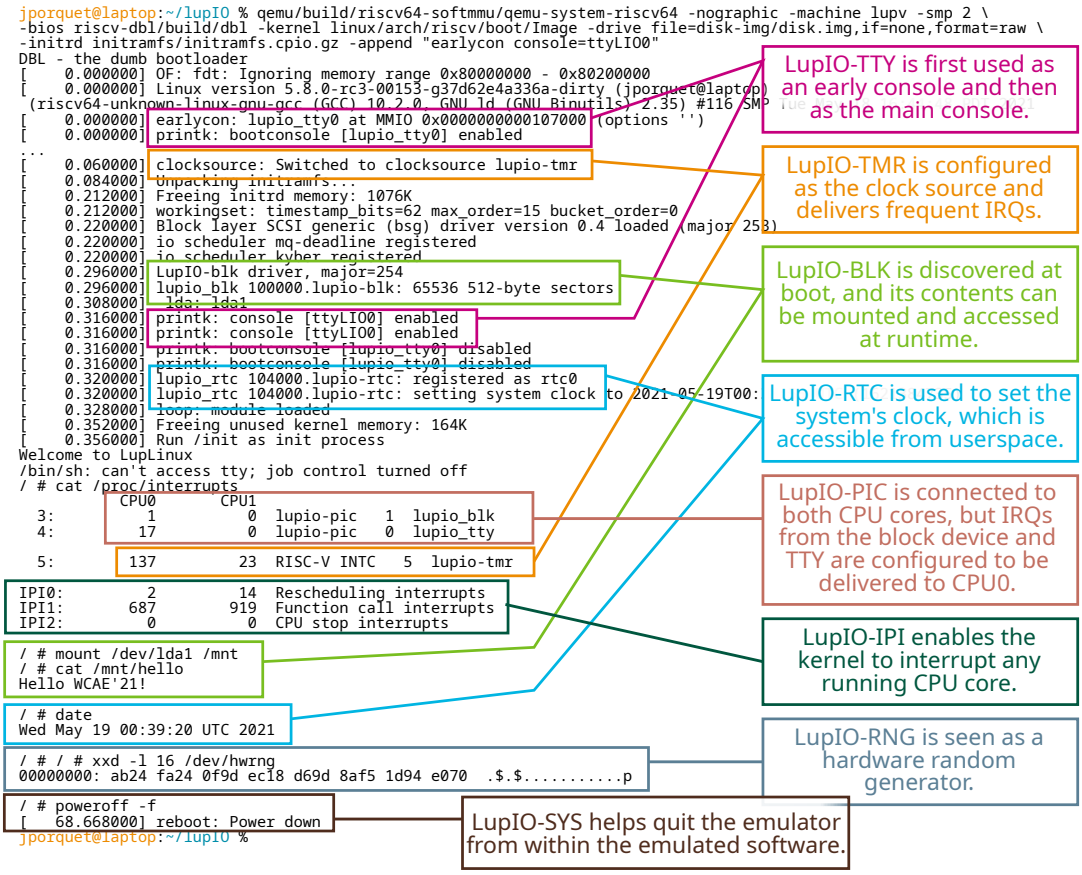

Figure 4: Linux equipped with LupIO device drivers running on a RISC-V based emulator equipped with LupIO virtual devices

6 POSSIBLE CURRICULUM INTEGRATIONS

The collection of LupIO devices can be used in many typical CS/CE curriculum courses, both hardware and software focused, and at both the undergraduate and graduate levels.

6.1 Computer organization

In typical computer organization courses, undergraduate students usually learn how to program in assembly language and explore the interactions between software and hardware. Students are generally directed to write assembly code that can access simple I/O devices, such as a terminal or a timer. A selection of LupIO devices is compatible with a gradual exploration of these interactions.

For example, the LupIO-TTY can be introduced to students in multiple steps. At first, they can start by simply transmitting characters to the first register, `WRIT`. Then, they can be exposed to the transmitter and the receiver, using polling to determine when the device is ready. Finally, they can write proper interrupt handlers in order to manage the device's events when enabling the IRQs.

Similarly, only the first register of the LupIO-TMR, `TIME`, can be shown to students at first, so that they can retrieve the value of the monotonic counter. Students can then interact with the entire device by actually configuring a timer and managing the timer's expiration by writing an interrupt handler.

6.2 Computer architecture

In a computer architecture course, students can be tasked with the actual implementation of a LupIO device. At the undergraduate level, they may be working with a simple device's specifications, such as LupIO-RNG, the random number generator. Depending on the exact course, it might be the software implementation of a virtual device, or the hardware implementation of a physical device using a hardware description language. At the graduate level, a more complex device's specifications may be used, such as LupIO-BLK, the block device.

6.3 Operating systems

In an OS course, instructors can show the implementation of an OS that runs on a system equipped with LupIO devices since the corresponding device drivers would be easy to understand. Since LupIO supports more complex SMP systems, it would also make an appropriate demonstration platform for more advanced graduate-level courses.

Students can otherwise be tasked with writing their own device driver for one of the LupIO devices. Depending on the level of the course, the device driver can be for a simple device, such as LupIO-RNG, or for a more complex device, such as LupIO-BLK.

7 CONCLUSION

In this paper, we presented LupIO, a comprehensive and open-source collection of education-friendly I/O devices. The current specifications, as well as the implementations of the QEMU virtual devices and Linux device drivers, are available on GitLab at https://gitlab.com/luplab/lupio.

In order to become a gold standard for educational I/O devices, the LupIO collection now needs to be widely advertised to the community. Increasing the visibility of this project should have two positive effects. First, receiving feedback and possible external contributions on these specifications would help us address unforeseen limitations or problems, and help us create a stable version. Second, it would promote the adoption of LupIO in other projects. We are already planning to work on the latter by developing models of LupIO in various education or research oriented projects such as SPIM, RARS, or gem5 [3].

REFERENCES

[1] Krste Asanović and David A Patterson. 2014. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146* (2014).

[2] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 46.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[4] Randal E Bryant and David R O'Hallaron. 2003. *Computer systems: a programmer's perspective*. Vol. 2. Prentice Hall Upper Saddle River.

[5] Alan Clements. 1999. Selecting a processor for teaching computer architecture. *Microprocessors and Microsystems* 23, 5 (1999), 281–290. https://doi.org/10.1016/S0141-9331(99)00049-6

[6] Alan Clements. 2010. ARMs for the poor: Selecting a processor for teaching computer architecture. In *2010 IEEE Frontiers in Education Conference (FIE)*. T3E–1–T3E–6. https://doi.org/10.1109/FIE.2010.5673541

[7] The OASIS Virtual I/O Device (VIRTIO) Technical Committee. 2019. *Virtual I/O Device (VIRTIO) Version 1.1.* https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf

[8] Google. 2007. *The Android Open Source Project.* https://android.googlesource.com/platform/external/qemu/+/master/docs/GOLDFISH-VIRTUAL-HARDWARE.TXT

[9] Neil AB Gray. 1986. *An introduction to computer systems.* Prentice-Hall, Inc.

[10] David A Holland, Ada T Lim, and Margo I Seltzer. 2002. A new instructional operating system. *ACM SIGCSE Bulletin* 34, 1 (2002), 111–115.

[11] Dimitris Kehagias. 2016. A survey of assignments in undergraduate computer architecture courses. *International Journal of Emerging Technologies in Learning (iJET)* 11, 06 (2016), 68–72.

[12] James Larus. 2005. Assemblers, linkers, and the SPIM simulator. *Appendix of Computer Organization and Design: the hardware/software interface, book by Hennessy and Patterson* (2005).

[13] National Semiconductor Corporation 1995. *PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs.* National Semiconductor Corporation. Datasheet.

[14] Linus Torvalds. 1991. *Linux.* https://kernel.org

[15] John Voelcker and Harold A. Hoeschen. 1986. Microprocessors: New architectures increase speed, and RISC processors are moving from university laboratories into commercial products. *IEEE Spectrum* 23, 1 (1986), 46–48. https://doi.org/10.1109/MSPEC.1986.6370962

[16] Kenneth Vollmar and Pete Sanderson. 2006. MARS: an education-oriented MIPS assembly language simulator. In *Proceedings of the 37th SIGCSE technical symposium on Computer science education.* 239–243.

[17] David Wheeler. 2001. *SLOCCount.* https://dwheeler.com/sloccount/