Operációs Rendszerek

AMP Laboratórium

VIMIAB00

"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with you using PREEMPT_RT."

Linus Torvalds

Bajczi Levente

Author: Levente Bajczi

AMP on Consumer SoCs

A System on a Chip (SoC) is a tightly integrated hardware element that contains most of the components necessary for the computer to function. Nowadays these incorporate mostly multi-core processors, as we will see during this laboratory exercise, where we will uncover the reasons why a general-purpose operating system is not the best idea for tasks where either exact timing or low and stable latencies are necessities.

Structure of the Exercises

The laboratory itself structures as follows:

1. Using a general-purpose kernel
    a. Measure latency with minimal stress on the cpus
    b. Measure latency with aggressive stress on the cpus
    c. Measure latency with aggressive stress and priority adjustments
        i. Low: 19
        ii. High: -20
    d. Measure latency while running in kernel-space
2. Using a real-time kernel
    a. Measure latency with aggressive stress and real-time priority
        i. 1
        ii. 99
    b. Measure latency with aggressive *real-time* stress and real-time priority
        i. Lower (e.g. 1 for the stress and 99 for the task)
        ii. Higher (e.g. 1 for the task and 99 for the stress)
3. Using an AMP system (Asynchronous Multi-Processing)
    a. Measure latency with minimal stress on the linux-cpus
    b. Measure latency with aggressive stress on the linux-cpus

Tasks

To achieve the measurements above, the following agenda is to be completed:

1. Write a program running on Linux that reads a GPIO and mirrors it to another (**3.1.**)
2. Write the same program inside a kernel module (using kthreads) (**4.1.**)
3. Compile (and flash) a PREEMPT_RT patched linux kernel for the board (**2.2.**)
4. Configure the board to use a bootloader (Das U-Boot) (**2.3.**)
5. Exclude the 4th core from the scheduler (**2.3.**)
6. Configure the kernel args so that only half of the RAM will show up as usable
7. Write a bare-metal program that does what the program in the first point does (**3.2.**)
8. Write a loader that manages the life cycle of the bare-metal CPU. (**4.2.**)

Finally, the measurements need to be carried out. Consult **Section 5.** to do them! You can decide if making a few measurements here-and-there works better for you, or you can do all the preparation and carry them out at the end. Allow approx. 20 mins for this section!

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Author: Levente Bajczi

Necessary Knowledge

General purpose tools

- SSH - SimpleWiki
- SCP - Wikipedia
- Makefiles - Wikipedia
- Mount - Wikipedia

Specific to this lab exercise (read through these before coming to the lab!):

- **AMP -** Wikipedia
- **PREEMPT_RT -** LinuxFoundationWiki
- **Linux Kernel Build System -** Linuxjournal

Useful commands

For some of these commands, *sudo* or being root is necessary.

- Make a file executable: `chmod +x <file>`
- Copy a file to the pi: `scp <local-file> pi@<ip>:<remote-file>`
  - The password is *raspberry*
- Log in to the pi using ssh: `ssh pi@<ip>`
- Launch a program: `./<program>`
  - Modify its 'nice' value: `nice -N ./<program> # for -20: nice -20`[1]
  - Modify its real-time priority: `chrt N ./<program>`[2]
- Extracting a *.tar.xz* file: `tar xvf <tar.xz file>`
- Check system information: `uname -a`
- List devices of a live system: `dtc -I fs /sys/firmware/devicetree/base`
- Print out the system log: `dmesg`
- Clear the system log: `dmesg -C`
- Flash an SD card with an image:
  `dd if=<in> of=<out> bs=4M status=progress conv=sync,noerror`
  - out: */dev/mmcblk0* or something similar
- Stress the CPUs: `stress-ng --cpu N -t Xm # N threads for X minutes`
  - Add *--aggressive* to crank the stress up
  - Add *sudo* to the beginning to make it even more aggressive (might lock up!)
  - More configuration: https://wiki.ubuntu.com/Kernel/Reference/stress-ng

Required Documentation

The repository includes a skeleton README.md, which should be filled out during this laboratory. Be concise, there is no need for lengthy paragraphs, but describe your work accurately. Advanced formatting is not required, but for simple modifications to the plain text use this cheat sheet when necessary: cheatsheet
Furthermore, all code (commented!) should be committed and pushed to the repository to make us able to check your work after the lab.

[1] https://askubuntu.com/a/656787/613336
[2] https://stackoverflow.com/a/52501811/4564311

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Author: Levente Bajczi

1. Setting up Cross-Compilation Toolchains

During this laboratory exercise, the following Raspberry Pi models are usable:

Model	SoC	µArch	Documentation	Base Address
RPi 2 B	BCM2836	ARMv7	Local peripherals Remote peripherals Datasheet	0x3F200000
RPi 3 B+	BCM2837	ARMv8		
RPi 4	BCM2711			0xFE200000

The toolchains for the different microarchitectures are available on the following links[3]:

- Bare-metal: GNU Toolchain | GNU-A Downloads
- Linux: Linaro Releases (*arm-linux-gnueabihf* or *armv8l-linux-gnueabihf*)

To use these toolchains, set up environment variables by placing the following lines into your .bashrc[4] file:

```
export CROSS_LINUX=<path-to-linux-toolchain>/bin/arm64-linux-gnueabihf-
export CROSS_BARE=<path-to-baremetal-toolchain>/bin/arm64-eabi-
```

To use these immediately, issue `source ~/.bashrc`.

To test the setup, and to experience the difference between the two, compile the following code using both `${CROSS_LINUX}gcc` and `${CROSS_BARE}gcc`.

```c
#include <stdio.h>
int main()
{
   printf("Hello World!\n");
   return 0;
}
```

Transfer to the RPi, modify its permissions to be executable, and launch the program.

Question: Did both of the toolchains work? Was this expected? Why?

[3] They are pre-downloaded into the Downloads folder in the home directory of the VM

[4] By e.g. issuing `nano ~/.bashrc` anywhere from a terminal. Use Ctrl+O to save, Ctrl+X to quit.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

2. Preparing the Images

To complete this laboratory exercise, you will need to prepare 3 different images:

1. Normal off-the-shelf Raspbian Lite
2. Raspbian Lite with the PREEMPT_RT patch enabled
3. Raspbian Lite with a modified kernel without access to the 4th core

Download the most up-to-date image from the official website[5], and make 2 copies of it:

1. normal.img
2. rt.img
3. amp.img

Place an empty and extensionless file named “ssh” in the boot partition of the image to enable SSH (you will need to mount the image to do this, see Section **2.1.**).

2.1. Mount an Image to Configure

To modify an image file, we will mount its partitions using an intermediary loop device (everything with *sudo*!):

1. `losetup -f #This will return with a usable loop device`
2. `losetup /dev/loop0 <img> #This will make the image use the device`
3. `fdisk -l /dev/loop0 #This will print out the partition info`
   a. You should see something similar:

```
Disk /dev/loop0: 2.21 GiB, 2369781760 bytes, 4628480 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x738a4d67

Device       Boot  Start     End Sectors  Size Id Type
/dev/loop0p1        8192  532479  524288  256M  c W95 FAT32 (LBA)
/dev/loop0p2      532480 4628479 4096000    2G 83 Linux
```

   Make note of the values in the *Start* column and the sector size (most likely 512b)!
4. `mkdir /mnt/boot; mkdir /mnt/rootfs`
5. `mount /dev/loop0 /mnt/boot -o offset=$((<sectorsize>*<start1>))`
6. `umount /mnt/boot`
7. `mount /dev/loop0 /mnt/rootfs -o offset=$((<sectorsize>*<start2>))`
8. `# do the necessary modifications, detailed in the next 2 sections`
9. `umount /mnt/rootfs`
10. `losetup -d /dev/loop0 # Un-register the device when done`

[5] It is pre-downloaded into the Downloads folder in the home directory of the VM.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

2.2. Replace the Kernel with a Real-Time Kernel

1. Get the sources: `git clone https://github.com/raspberrypi/linux.git --depth 1 --branch rpi-4.19.y-rt`
2. Depending on the Pi's model, issue either:
   a. `make bcm2709_defconfig ARCH=arm CROSS_COMPILE=$CROSS_LINUX (Pi 2 B, Pi 3B+)`
   b. `make bcm2711_defconfig ARCH=arm CROSS_COMPILE=$CROSS_LINUX (Pi 4)`
3. `make -j4 zImage modules dtbs ARCH=arm CROSS_COMPILE=$CROSS_LINUX` *This will take a long time. Take a short break while it compiles, maybe read up on AMP systems[6] or the PREEMPT_RT patch[7]!*
4. Make sure that you have mounted the partitions of the image being modified to /mnt/{boot,rootfs} (one at a time)
5. Delete the following files in */mnt/boot*: **.dtb, kernel*, overlays/**
6. `make modules_install INSTALL_MOD_PATH=/mnt/rootfs ARCH=arm CROSS_COMPILE=$CROSS_LINUX`
7. `cp arch/arm/boot/zImage /mnt/boot/`
8. `cp arch/arm/boot/dts/bcm***.dtb /mnt/boot/ #***: depends on the Pi model. The possibilities are:`
   a. `2711-rpi-4-b`
   b. `2710-rpi-3-b-plus`
   c. `2709-rpi-2-b`
9. `cp arch/arm/boot/dts/overlays/*.dtb* /mnt/boot/overlays/`
10. Put `` `kernel=zImage` `` at the end of /mnt/boot/config.txt
11. Flash the image using balenaEtcher or *dd* and try to boot the new kernel.

Question:
1. **What is the name of the real-time kernel? What release and version is it?**
2. **What devices are present in the booted system as part of the device tree?**

[6] https://en.wikipedia.org/wiki/Asymmetric_multiprocessing
[7] https://wiki.linuxfoundation.org/realtime/documentation/start

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

2.3. Replace the Bootloader for the AMP-Setup

The default bootloader (which runs on the GPU of the Pi) does not respect all kernel parameters that are passed to it. Therefore, we need to insert another 2nd stage bootloader into the boot sequence, which is called *Das U-Boot*[8].

1. Clone the repo: `git clone git://git.denx.de/u-boot.git`
2. Configure: `make rpi2_defconfig` # even for 3b+
3. Build: `make u-boot.bin -j4`
4. Mount the *boot* partition of the image being modified to */mnt/boot*
5. Copy the created *u-boot.bin* file to the boot partition
6. Create a file called *boot.cmd* in the boot partition containing the following code:

```
setenv bootargs earlyprintk console=tty0 console=ttyAM0
root=/dev/mmcblk0p2 rootfstype=ext4 rootwait noinitrd
fatload mmc 0:1 ${fdt_addr_r} bcm2710-rpi-3-b.dtb
fatload mmc 0:1 ${kernel_addr_r} kernel7.img
bootz ${kernel_addr_r} - ${fdt_addr_r}
```

7. Issue `mkimage -A arm -O linux -T script -C none -n boot.scr -d boot.cmd boot.scr`
8. Verify that a *boot.scr* has been created in /mnt/boot
9. Add the following line to *config.txt*: `kernel=u-boot.bin`

If done well, this process has inserted the U-Boot bootloader into the boot process seamlessly, meaning that the same kernel works as before but with an additional step when booting.
To achieve our goal of not having all the cores under linux's scheduler, we have to configure the kernel parameters the bootloader passes to the booting kernel. This is specified in the *bootargs* environment variable in the *boot.cmd* file. A (somewhat complete) list of parameters can be found in the kernel's documentation[9].

Question:

1. **Which parameter has to be set to only allow 3 CPUs in the system?**
2. **Which parameter can be used to limit the RAM usage of the system to 512M?**
3. **What do the existing parameters mean in the *boot.cmd* file?**

Modify the boot.cmd file (and of course run through steps 7-8) in the boot partition. Flash the SD card and boot the pi, then verify that it indeed only has 3 cores and 512M of RAM[10].

[8] https://www.denx.de/wiki/U-Boot
[9] https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html
[10] Use the *htop* CLI program to verify this.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

3. Using the GPIO pins on the Raspberry Pi

In the Remote Peripherals Datasheet above, there is a picture and a short description of the memory mapping of the Raspberry Pi's SoC under section **1.2**.

Question:
1. **At which bus address does the I/O block start?**
2. **Where can we reach it from software running *without* an OS?**
3. **Where can we reach it from software running *on top of* an OS running in kernel mode?**

In the same document, we can find the registers used to configure the General Purpose Input-Output pins of the SoC (GPIO) under section **6**. We know that to use a GPIO we need to do the following tasks:

- Configure the pin as input/output (registers named **GPFSELn**)
- Write to a SET register when using it as an output to pull it **up** (**GPSETn**)
- Write to a CLEAR register to pull it **down** (**GPCLRn**)
- Read a LEVEL register when using it as an input to read the current state. (**GPLEVn**)

Question:
1. **Which register (name and *bus* address) is responsible for selecting the GPIO function of pins #14, #15 and #16? Which bits?**
2. **Which registers are the corresponding SET, CLEAR and LEVEL registers (name and *bus* address)?**

3.1. Mapping Physical Addresses to Virtual Addresses

Even though while running in kernel mode, the I/O is reachable as a simple I/O instruction, user mode is not so lucky. Instead of that, the use of the *mmap* function is necessary. The arguments to be passed are the following:

1. Addr - NULL, because we don't care where it will end up. The kernel knows better.
2. Prot - `PROT_WRITE`
3. Flags - `MAP_SHARED`
4. Fd - Returned file descriptor of the file "**/dev/gpiomem**" using the open system call. Use `O_RDWR | O_SYNC` as the flags for the *open* syscall.
5. Offset - 0, because the file above is aligned to the start of the GPIO block
6. Size - calculate it using the value of the GPIO register at the highest address

Never forget to close and munmap after being done using the GPIO!

Task: Write Task #1! Use pins 14->15 on the RPi 4, 15->16 otherwise!

Before running the task, make sure a supervisor checks your set up (so you can make sure that the RasPi will live). If you would rather continue with further setup, you can proceed with the instructions on the next page.

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

3.2. Running Bare-Metal Programs on the Pi

Bare-metal in this context means that there is no operating system we could rely on while developing the program we are trying to run. This means that anything we want to achieve has to be done completely from scratch, there is no abstraction of the hardware we could use. As already uncovered in Section 1., not even *printf* has a default implementation - the compiler does not know *how* it could give us an output.
Another difference arises from the memory usage - unless we explicitly say otherwise, the program has direct access to the entirety of the *physical* addresses. This makes it easier to develop applications, but at the same time, makes it a security vulnerability to do so. In our use-case, the cores running Linux will see 512MBytes of the RAM and the rest (however large it is) will belong to the bare-metal program. It could, however, potentially see and even modify the RAM contents of the processes running on top of Linux.
Furthermore, the core is *uninitialized* the first time we make it jump to the program's address, and therefore we need to set up a few things before we could un anything else. To do so, we will need the following file structure:

- **Makefile** - to facilitate compilation
- **main.c** - contains our program's code
- **start.S** - contains necessary code to bootstrap the CPU and provide basic LCM[11]

The skeleton is available in the folder *task7-gpio-mirror-baremetal*.

Task: Write Task #7! Use pins 14->15 on the RPi 4, 15->16 otherwise!
Using *${CROSS_BARE}nm metal.elf* determine where each of the functions are placed by the linker. What is the entry point's address?

3.3. Starting a Core on the Raspberry Pi

Normally, the boot process takes care of starting all cores a given system has, but in this case, we have specifically made it so that it only has access to 3 of the 4 available cores. We *could* start it from under Linux, but then it would be assigned tasks from the scheduler, and we want to refrain from that. The easiest way of figuring out how to wake up a core is to look at the kernel's source and see how it does it[12]. In this case, the cores are in a *Wait-for-Instruction* (WFI) loop, meaning they are monitoring a register which will contain a memory location to jump to. In the case of the Raspberry Pi, this register is the *Mailbox* of the core, more specifically the 3rd one.

Question: Using the Pi's local peripheral documentation[13], figure out the following:

1. **What is the *SET* address of the 3rd core's 3rd mailbox?**
2. **What is the CLEAR address of the 3rd core's 3rd mailbox?**
3. **Which is readable?**

[11] Life Control Management - Describing what should happen before/after the program's end/start
[12] https://github.com/raspberrypi/linux/blob/rpi-4.1.y/arch/arm/mach-bcm2709/bcm2709.c#L1248
[13] https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

4. Linux Kernel Modules

Kernel modules are pieces of code that can be loaded and unloaded into the kernel upon demand. They extend the functionality of the kernel without the need to reboot the system. In the scope of this project we will create two kernel modules, as Tasks #2 and #8 outlined. The Linux Kernel Build System[14] makes the development and compilation of a kernel module very convenient, but we must abide by a few rules:

- The running kernel's source (more specifically, the headers) must be used for compiling the module
- To build the module (from the same directory as the source.c):
  `make obj-m+=<source.c without .c >.o ARCH=arm \`
  `CROSS_COMPILE=$CROSS_LINUX -C <linux-source-root> M=$PWD modules`
- To clean the project (from the same directory):
  `make -C <linux-source-root> M=$PWD modules`

This can easily be incorporated into a Makefile of its own:

```
obj-m += kernel-module.o
all:
   make ARCH=arm -C ../../linux M=$(PWD) modules
clean:
   make ARCH=arm -C ../../linux M=$(PWD) clean
```

The generic outline of any kernel module (my_module.c) is the following:

```
#include <linux/module.h> // Necessary include for kernel modules
MODULE_LICENSE("GPL");    // Could be any other license as well
MODULE_AUTHOR("Author Name");         // Your name
MODULE_DESCRIPTION("Short description"); // Concise summary
MODULE_VERSION("0.01");               // Version number
static int __init lkm_example_init(void) // This will run when loaded
{ printk(KERN_INFO "Hello World!"); return 0; }
static void __exit lkm_example_exit(void)// This will run when unloaded
{ printk(KERN_INFO "Goodbye World!"); }
module_init(lkm_example_init); // To make the correct function run
module_exit(lkm_example_exit); // To make the correct function run
```

After compiling the module, a *.ko* will appear in the directory of the source, along with a plethora of other (for us) uninteresting files. After copying it to the device, you can load and unload it with *insmod <file>.ko* and *rmmod <file>.ko*. Check the log (*dmesg*) to see what happens! You should see the messages "Hello World" and "Goodbye World" in the log.

Question: How long was the time difference between two 'printk' statements?

[14] https://www.linuxjournal.com/content/kbuild-linux-kernel-build-system

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

4.1. Kthreads

In the previous section, the basic skeleton of a kernel module was introduced. It did something when it was loaded and something else when it was unloaded - but what if some long-running functionality was necessary? In our case, this is the situation, as we want to respond to GPIO signals right away *at any time* - meaning it should not be restricted to the time we load it. For this purpose, two solution exist:

1. Making the long-running functionality time-constrained, i.e. limiting how long it should be allowed to run. Then it can be invoked from the init function.
2. Creating a long-running background thread that is allowed to exist until we unload the module.

Option #2 is a lot more elegant, and therefore we will explore that one.
As we are running in kernel mode, there is no way to just simply invoke a *fork* syscall and use it to run a function asynchronously. Instead, since kernel version 2.6, there are new primitives to be used for this purpose (until then, this functionality was always implemented in a custom way), called kthreads, which are lightweight processes running in kernel mode.
Its use is fairly simple and well detailed in the source (https://lwn.net/Articles/65178/).

Task: Write Task #2 based on Task #1 and #7! You can either use ioremap[15] for the physical address or simply the already mapped virtual address.

4.2. Character Devices[16]

So far we have seen how kernel modules are built - but how can we actually use them, besides loading and unloading them? The UNIX philosophy (and by extension, that of Linux) says that *everything is a file*[17] - and this is how most device drivers are built. For example, there are files under */dev/* representing physical disks such as HDDs and SSDs (e.g. /dev/sda) and virtual terminals (e.g. /dev/tty1). In this example, the former is called a *block special* and the latter a *character special* file (confirm these by issuing *file /dev/{sda,tty1}*).
To achieve the goals of Task #8, we are going to use a *character special* file which will provide the following services:

- When being read, it prints a message to the kernel log stating it is write only
- When being written to, it treats it as a program to be run - if the last program has terminated and is currently waiting for the next, the module will load the new program into memory and then make the core jump to it.

Linux provides us with a well-build toolset to deal with character devices[18]. The skeleton is available in the folder *task8-loader* and includes further tasks.

Task: Write Task #8 based on the comments in the source!

[15] https://learnlinuxconcepts.blogspot.com/2014/10/what-is-ioremap.html
[16] https://linux-kernel-labs.github.io/master/labs/device_drivers.html
[17] https://en.wikipedia.org/wiki/Everything_is_a_file
[18] http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.

Author: Levente Bajczi

5. Measuring Latency

We are going to use the Logsys FPGA cards to test the latency of the 3 different setups we have prepared. Writing the test code (in Verilog) is *not* in the scope of this exercise, but it can be found in the prepared repository if you are interested in its inner workings. To use it, you will need to flash it onto the device using the *logsys-test* program.

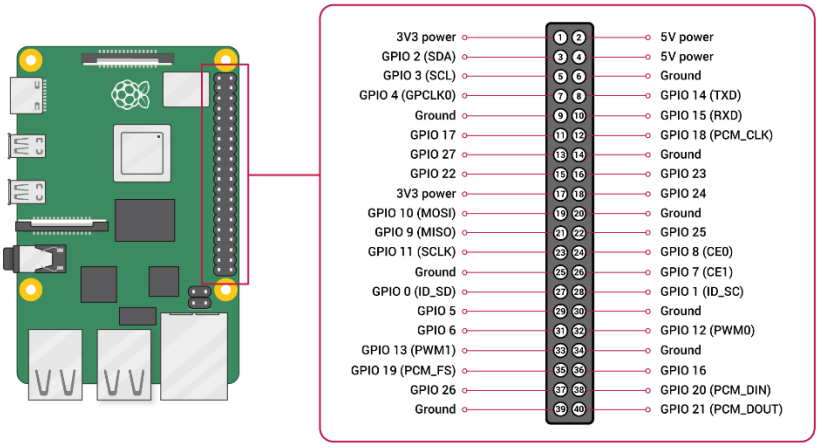
1. Run `echo $PWD/<bitfile>` to find out the absolute path of the bitfile
2. Run `logsys-test`
3. In its console, write `vcc on`
4. Write `conf bit <absolute-path-to-bitfile>`

If everything was done correctly, the 7-segment display of the FPGA spells *DEAD*, and led0 is lit up. In order to carry out a latency measurement, connect pin 13 to the input GPIO and pin 14 to the output GPIO of the Pi. Use headers *A* of the card.
The pinout of the FPGA headers:

(15) Input	(13) I/O	(11) I/O	(9) I/O	(7) I/O	(5) I/O	(3) +3,3V	(1) GND
(16) Input	(14) I/O	(12) I/O	(10) I/O	(8) I/O	(6) I/O	(4) I/O	(2) +5V

9-1. ábra: A bővítőcsatlakozók lábkiosztása.

The pinout of the Pi:



Press *btn0* to start a measurement. When the 7-segment display is lit up again (cca. 2s), it will display the *average* latency (this is denoted by the lit up ld1). From here on, do the following:

1. Press btn1 to display the average
2. Press btn2 to display the minimum
3. Press btn3 to display the maximum

All values are in *cycles* of the 16.0000MHz clock the FPGA is equipped with.

Question & Task:

1. **How long is *one cycle* of the fpga in (nano)seconds?**
2. **Measure latencies using the laid-out measurement plan of the first page! Place your findings in a spreadsheet. Do each measurement at least 5 times!**

Supported By The ÚNKP-19-1 New National Excellence Program Of The Ministry For Innovation And Technology.