CSE 237A Wi'16: Individual Project Part 1 Detailed Instructions

Environment setup and workload characterization

During the first part of the project, you will setup the development environment for RPi2, and then measure its performance. The purpose of this document is to help you to prepare your system for the project and measure performance counters by:

- Installing the required tools
- Downloading and compiling the kernel source code
- Cross-compiling a sample C program and a kernel module
- Enabling PMU and ensuring you can read/write values from registers
- Analyzing the performance of the provided workload

Hardware and software requirements

- Raspberry PI 2
- Intel ISA, 64 bit machine (x86_64)
- Ubuntu 64bit 14.04.3 LTS desktop edition
    - You may run it as a virtual machine in VirtualBox.
    - For package installation commands, root access through sudo command is required.
    - If you are familiar with Linux and already have other Linux version (e.g., Ubuntu 12.04 LTS), you might get it to by using additional repos/packages, however, we cannot support that.
- Cross-compiler

Install Ubuntu

You may install Ubuntu 14.04 directly on your machine ("bare metal") or as a virtual machine (VM). If you are NOT familiar with Linux environment, we highly recommend using a VM. If you are already familiar with Linux and have your own Linux machine, we may use it instead.

Bare metal instructions (for Windows):

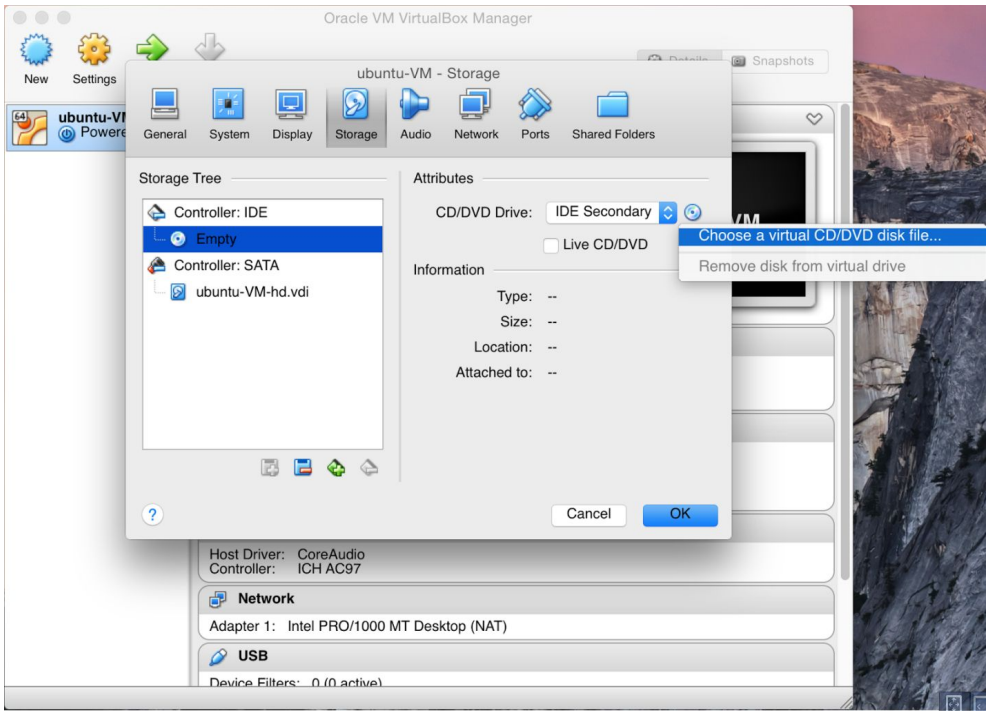
- Create a bootable USB stick:
    - http://www.ubuntu.com/download/help/createausbstickonwindows
- Follow official installation instructions:
    - http://www.ubuntu.com/download/desktop/installubuntudesktop

Dual booting Linux and OS X is not supported as a part of this class.

VM instructions (Mac or Windows):

1. Download and install VirtualBox:
    - https://www.virtualbox.org/wiki/Downloads
2. Download Ubuntu 14.04:
    - From http://releases.ubuntu.com/14.04/, download "64bit PC (AMD64) desktop image" (a large ISO image almost 1GB)
3. Open VirtualBox

4. Click Machine > New…
5. Enter the following:
    - Name: ubuntuVM
    - Type: Linux
    - Version: Ubuntu (64bit)
6. Set a proper RAM size. For example, if your host machine has 4GB+ RAM, you may allocate 1GB to the virtual machine
7. Create a new virtual hard drive with “VDI format” and “Dynamically allocated” options. Set the disk size by more than 16 GB.
8. Leave other settings to their default values unless you know what you're doing.
9. Once the VM has been created, right click > "Settings" > "Storage"



10. Add the downloaded ISO image as a virtual disk file and click "OK"
11. Start the VM
12. Select "Install Ubuntu"
13. Take default options (including "Erase disk and install Ubuntu" this refers to the virtual disk)
14. After installation completes, repeat Step 8 and remove the ISO file (if any) so that the IDE controller list reads "empty" again.
15. Restart as prompted by the installer

Install Guest Additions (optional)

Guest Additions support many nice features for Ubuntu in VM, including a solution for screen resolution issue. To install the guest additions, click “Devices” > “Insert Guest Additions CD image” in the menu of the virtual machine window, and follow its install instruction. You need to reboot the VM after installing it to apply the changes.

If the default CD image for Guest Additions is not completely installed, you may try to install it from Debian packages instead. Open a terminal using CTRL+ALT+T in your VM, and then type:
$ sudo apt-get install virtualbox-guest-dkms virtualbox-guest-utils virtualbox-guest-x11

If it gives you a message: “unmet dependency”, you may try the following:
$ sudo apt-get remove libcheese-gtk23
$ sudo apt-get install xserver-xorg-core
$ sudo apt-get install virtualbox-guest-dkms virtualbox-guest-utils virtualbox-guest-x11

Troubleshooting

If any step quits prematurely and shows an error message, log the error and read it. Some common errors can be figured out with a quick web search or by posting to class piazza page. To help us debug the problem, include the error message and output of the following commands (include packages.txt as an attachment)

- uname-a
- lsb_release-a
- echo $PATH
- dpkg --get-selections > ~/packages.txt

Kernel Building and Installing for RPi2 on Your Machine

Refer to the link on the official RP website:
https://www.raspberrypi.org/documentation/linux/kernel/building.md
Parts of this instruction manual are written based on these official guidelines.

Step 0: Install git and curl

The download of the tool chain and source codes requires ***git***. You can also download files (e.g., the sample C code provided on the class website) from the Internet using ***curl***. Using the root privilege, you can download them as follows:
$ sudo apt-get install git curl
You can download any required libraries and linux programs in a similar way. For example:
$ sudo apt-get install <ANY_PROGRAM_OR_LIBRARY>)

Step 1: Install toolchain

To compile the Linux kernel for RPi2 on your machine (i.e., bare metal or VM), you require a suitable Linux cross-compilation toolchain - i.e. a compilation suite to make executable binaries for a different ISA from your host machine.

1. Get tool chains by using the following command:
   ~$ mkdir ~/RPdev
   ~$ cd ~/RPdev
   ~/RPdev$ git clone https://github.com/raspberrypi/tools
2. Put the toolchain path to your PATH (to make it easier for later command lines)
   ~/RPdev$ export
   PATH=$PATH:~/RPdev/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin
   (You may put the line in your ~/.bashrc to make it permanent.)

Step 2: Download kernel source

The kernel source for this project is customized for the low-level performance counter access we need. You can clone the repository using git as follows:

~/RPdev$ git clone http://seelab.ucsd.edu/~shepherd/cse237a_rpikernel.git linux

Now, the source code is downloaded in the "~/RPdev/linux" directory.

Step 3: Build kernel source

To build the sources for cross-compilation, there may be extra dependencies (depending on your ubuntu version) beyond those you've installed by default with Ubuntu.
Enter the following commands to build the sources and Device Tree files.

~/RPdev$ cd linux
~/RPdev/linux$ KERNEL=kernel7
~/RPdev/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- bcm2709_defconfig
~/RPdev/linux$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs

To speed up compilation on multiprocessor systems, and get some improvement on single processor ones, use -j n where n is number of processors * 1.5. For example,
$ make -j4 ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs

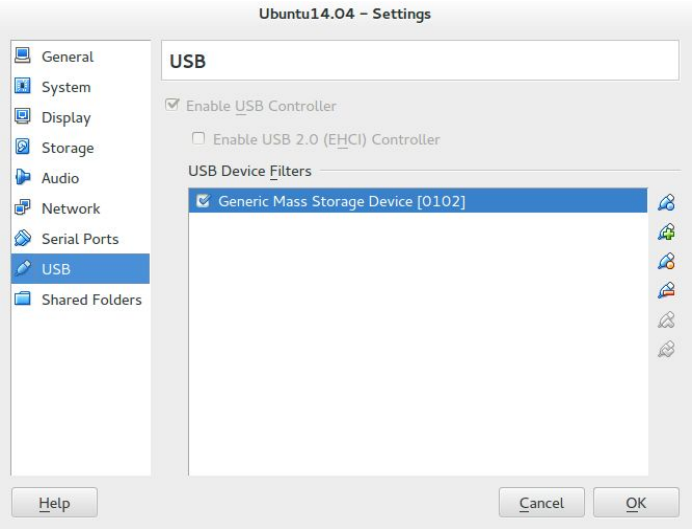
Step 4: Check the compiled kernel image
To check the compiled kernel images, use the following commands and compare results:

~/RPdev/linux$ file arch/arm/boot/zImage
arch/arm/boot/zImage: Linux kernel ARM boot executable zImage (little-endian)
~/RPdev/linux$ file vmlinux
vmlinux: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, BuildID[sha1]=73ba8a1e58c20f55094ac0442256a10f23be2c29, not stripped

The hash value might be different from what you get.

Step 5: Install kernel onto the SD CARD
Install the compiled kernel onto the RPi2. Take your microSD card and plug it into your machine using a SD card reader or a built-in SD card slot.

- If you're using VM, you need to forward the SD card reader to VM. The device names shown in the images below may vary from machine to machine.
- In most case for VirtualBox with a USB SD card reader, the instructions are as follows:
  1. Plug off the SD card from the reader
  2. Turn off your VM (i.e., the installed Ubuntu 14.04)
  3. Open the Virtualbox application (not the VM)
  4. In Settings menu, go to USB section
  5. Add a USB filter for the USB SD card reader
  (the "+" button on the right side of the dialog)



6. Close the dialog, and start the VM again

7. Once booting is completed, plug the SD card in the reader
8. Type the following command in the terminal to make sure all partitions are loaded
   $ sudo partprobe

- If you're using the SD card slot of a recent MacBook Pro, you may refer to: http://superuser.com/questions/373463/how-to-access-an-sd-card-from-a-virtual-machine

Once the SD card is properly detected, check the partitions as follows:

```
~/RPdev$ lsblk
sdb      8:32   1   7.4G  0 disk
├─sdb1   8:33   1 814.3M  0 part
├─sdb2   8:34   1     1K  0 part
├─sdb3   8:35   1    32M  0 part
├─sdb5   8:37   1    60M  0 part
└─sdb6   8:38   1   6.5G  0 part
```

These partition names might be different from what you get depending on your VM or machine. In the above case, it shows the partitions of a purchased NOOBS card. **sdb5** and **sdb6** are the target partitions that the kernel will be installed.

If you have any other SD Card, first prep it as follows using these instructions (but flash it with Raspbian, NOT NOOBS). In this case, you will see only **sdX1** and **sdX2**. Mount these as appropriate in place of **sdb5, sdb6**, respectively, below.

Mount the PROPER partitions (highlighted as bold text) as follows:

```
~/RPdev$ mkdir mnt
~/RPdev$ mkdir mnt/fat32
~/RPdev$ mkdir mnt/ext4
~/RPdev$ sudo mount /dev/sdb5 mnt/fat32
~/RPdev$ sudo mount /dev/sdb6 mnt/ext4
```

Next, install the compiled modules and the new kernel and make it bootable.

```
~/RPdev$ cd linux
~/RPdev/linux$ sudo env "PATH=$PATH" make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
INSTALL_MOD_PATH=../mnt/ext4 modules_install
~/RPdev/linux$ sudo scripts/mkknlimg arch/arm/boot/zImage ../mnt/fat32/kernel-41.img
~/RPdev/linux$ sudo cp arch/arm/boot/dts/*.dtb ../mnt/fat32/
~/RPdev/linux$ sudo cp arch/arm/boot/dts/overlays/*.dtb* ../mnt/fat32/overlays/
~/RPdev/linux$ sudo cp arch/arm/boot/dts/overlays/README ../mnt/fat32/overlays/
```

Then, change the configuration so that the system uses the installed “kernel-41.img”. Edit “../mnt/fat32/config.txt” using a text editor, e.g., vi or nano or gedit. You need a root privilege. For example, if you’re using vi,
~/RPdev/linux$ sudo vi ../mnt/fat32/config.txt

In the opened file, replace the line “kernel=kernel.img” with the line “kernel=kernel-41.img”.

Finally, unmount all the partitions.
~/RPdev/linux$ sudo umount ../mnt/fat32
~/RPdev/linux$ sudo umount ../mnt/ext4

Now, you can plug the microSD card into RPi2 and turn on its power!
In order to check if the OS boot was successful, type the following in the RP2 to check the build date. It should show the date and time when you compiled the kernel.

$ uname -a
Linux raspberrypi 4.1.13-v7+ #1 SMP PREEMPT <BUILD_DATE_TIME> armv71 GNU/Linux

Cross-compiling C code

Step 1: Download the sample C code (provided on the class website) into "~/RPdev/test" directory.

```
~/RPdev$ mkdir test
~/RPdev$ cd test
~/RPdev/test$ curl -O http://cseweb.ucsd.edu/classes/wi16/cse237A-a/project/proj1/test.c
```

Step 2: Cross compile the given source code for the RPi2 target.

```
~/RPdev/test$ arm-linux-gnueabihf-gcc -o test.out test.c -static
```

The result should be executable: "test.out".

Step 3: Check if the cross-compiling was successful by looking at the file type of test.out:

```
~/RPdev/test$ file test.out
test.out: ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV), statically linked, for GNU/Linux 2.6.26, BuildID[sha1]=462b26711b7c38de9a653c4371597b57018c3531, not stripped
```

The description of "ELF 32-bit LSB executable" and "ARM" indicates that your binary is ready to be executed on the ARM processors of RPi2.

Step 4: Check if it really works on RPi2

First, copy the file to RPi2 using either a USB pen drive or Internet. For example, if you're using a USB pen drive, you have to mount the USB drive after connecting it:

```
$ sudo mkdir /media/usbdrive
$ sudo mount /dev/sda1 /media/usbdrive
$ cp /media/usbdrive/test.out ~/
```

Then, execute the file:

```
$ cd ~/
$ chmod a+x test.out
$ ./test.out
Hello RP World!
```

Practice kernel module

A kernel module is an object file that contains procedures to extend the running kernel. You need to develop a kernel module in order to read performance counters. If you are NOT familiar with kernel module, we highly recommend visiting the following link: Linux Kernel Module Programming Guide: http://www.tldp.org/LDP/lkmpg/2.4/html/c147.htm

Let's start with a simple kernel module. Use your installed VM to run this example.

1. **Start with simple kernel module: hello world (host compile)**

- Create a kernel module file: hello.c

```
/*
 *  hello.c - The simplest kernel module.
 */
#include <linux/module.h>   /* Needed by all modules */
#include <linux/kernel.h>   /* Needed for KERN_INFO */

int init_module(void)
{
   printk(KERN_INFO "CSE237A: Hello world.\n");
   return 0;
}

void cleanup_module(void)
{
   printk(KERN_INFO "CSE237A: Goodbye world.\n");
}
```

- Generate a Makefile

```
obj-m += hello.o
PWD := $(shell pwd)

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

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

Build the kernel module using the Makefile.
$ make

Check module information with the following command:

$ modinfo hello.ko

This kernel module can run only on this host architecture, but not on the Raspberry Pi's architecture (ARMv7). The vermagic prefix (3.8.0-44 in the example below) is the kernel version of your machine. This will vary depending on your machine configuration.

```
$ modinfo hello.ko
filename:       hello.ko
srcversion:     140276773A3090F6F33891F
depends:
vermagic:       3.8.0-44-generic SMP mod_unload modversions
```

Now load this kernel module. To load your kernel module, use the command "insmod" with root privilege (sudo). To unload your kernel module, use "rmmod".

$ sudo insmod hello.ko ← *load hello.ko*
$ sudo rmmod hello ← *unload hello.ko kernel module*

You can check the result of the kernel module with "dmesg" command. The function "printk" generates a log for the kernel module. Thus, you can see "CSE237A: Hello world." and "CSE237A: goodbye world." messages with dmesg command. To see the last log lines, use "tail" command as follows:

```
$ sudo insmod hello.ko
$ sudo rmmod hello
$ dmesg | tail -2
[178474.908145] Hello world.
[178479.783346] Goodbye world.
```

2. Cross-compile hello world kernel module for Raspberry Pi

gcc is the compiler on the host machine specific to its architecture. RPi2 is an ARM-based machine. You need to specify the architecture type and the ARM-based compiler from the previously-downloaded toolchain to use the Makefile. Create a new directory and copy the hello.c file there. Then, write the following Makefile in that same directory. (You have to give a proper path for KDIR.)

```
obj-m := hello.o
KDIR := /home/YOUR_ACCOUNT/RPdev/linux   # Your kernel source directory
PWD := $(shell pwd)

default:
	$(MAKE) -C $(KDIR) M=$(PWD) ARCH=$(ARCH)
CROSS_COMPILE=$(CROSS_COMPILE) modules

clean:
	$(MAKE) -C $(KDIR) M=$(PWD) ARCH=$(ARCH) clean
```

Then, run make to generate a kernel module for the appropriate target architecture and compiler.

```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
```

This will generate hello.ko kernel module. Check the module information. This module is compatible with ARMv7 and is compiled using the provided Linux kernel source for the Raspberry Pi 2. Thus, its kernel version is 4.1.13.

```
$ modinfo hello.ko
```

```
$ modinfo hello.ko
filename:       hello.ko
srcversion:     140276773A3090F6F33891F
depends:
vermagic:       4.1.13-v7+ SMP preempt mod_unload modversions ARMv7
```

Copy the kernel module (“hello.ko”) to RPi2. Use ssh or or copy to a drive. Then, check the kernel number on RPi2.

```
$ uname -a
```

If you correctly finish the custom kernel installation, you should see the following message.

```
Linux raspberrypi 4.1.13-v7+ #1 SMP PREEMPT Tue Dec 8 00:14:48 PST 2015 armv7l GNU/Linux
```

Compare the kernel version with that of hello.ko. Those two should be the same. You downloaded/built the kernel source and installed it onto your RPi2. This means that both the running kernel and the kernel source have the same configuration. Thus, hello.ko should run without any errors.
Test your install on RPi2 by installing and uninstalling the kernel module to get the messages shown in the box below.

```
$ sudo insmod hello.ko
$ sudo rmmod hello
$ dmesg | tail -2
```

```
$ dmesg | tail -2
[  249.740357] CSE237A: Hello world.
[  273.468906] CSE237A: Goodbye world.
```

If you see errors during the install/uninstall process, it may come from the kernel compatibility issue. Even though the running kernel and the kernel source have the same numeric value, the kernel module may not be loaded if the two are using different configuration options. This may result in invalid module format error.

- You can check the kernel source version on your VM that you compiled using the kernel image file (vmlinux). Move to the kernel source directory and type
  $ make kernelversion
  4.1.13
- The version must be same to that of the running kernel of RPi2. ("uname -a" command)
- If those number are the same, it means that you didn't successfully install the kernel image. Then, carefully follow the installation instruction again, and reinstall kernel image. You may refer to the official website:
  https://www.raspberrypi.org/documentation/linux/kernel/building.md.

Setup the system to use only a single core

Change the core usage settings so that your RPi2 uses only a single core (out of 4 cores of quad-core Cortex-A7 of RPi2) for this project. This allows us to execute programs on a single core and accurately measure performance of that core. Having multiple cores could offload some of the computation and throw off our PMU measurements. To do this, follow this:

- add "maxcpus=1" in /boot/cmdline.txt
  - e.g., "dwc_otg, …. rootwait maxcpus=1"
- reboot RPi2
  - e.g., $ sudo reboot
- Go /sys/devices/system/cpu and read two files: "offline" and "online".
  Each file shows unavailable and available CPUs.
  $ cat /sys/devices/system/cpu/online → should be 0
  $ cat /sys/devices/system/offline → should be 1-3

Now your RPi2 uses a single core.

Change overclocking setting

Next turn on the overclocking to ensure that you get a wide range of frequencies:

- Type in the terminal

  $ sudo raspi-config
- Enter “7 Overclock” menu
- Choose “Medium 900MHz ARM, 250 core, 450 MHz SDRAM, 2 overvolt”
- In the main menu, check “Finish” and reboot.

Now, your RPi2 can run at 900MHz (and 600MHz).

Prepare for you Checkpoint Demo

- Make sure that your RPi2 is using a single core with the medium overclocking setting and can run your sample kernel module.
- Download the following program binary onto your RPi2 before coming to the demo: http://cseweb.ucsd.edu/classes/wi16/cse237A-a/project/proj1/check_kernel
    - Check if the program works correctly:
      $ sudo ./check_kernel
      SUCCESS
        - If it prints "FAIL", your kernel has not been updated correctly. Please make sure that you update your RPi2 using the provided kernel source.
        - If it prints nothing on the terminal, reboot your system.
- Bring your RPi2 with the compiled/tested sample kernel module to cse 3219 on Jan 14 for us to test.

Individual Project Part 1 - Performance Monitoring Unit (PMU)

The processor (ARM v7-Cortex 7) includes logic to gather various statistics on the operation of the processor and memory system during runtime, based on the PMUv2 architecture. These events provide useful information about the behavior of the processor that you can use when debugging or profiling code. The processor PMU provides four event counters. Each event counter can measure any of the events available in the processor. Thus, you can access the following counter registers:

- Four 32-bit counters that increment when the corresponding event occurs.
- One cycle counter that increments based on the processor clock.

Access the PMU using "c9 register". The following table shows the 32-bit wide system control registers.

Table 4.10. c9 register summary

CRn	Op1	CRm	Op2	Name	Reset	Description
c9	0	c12	0	PMCR	0x41072000	*Performance Monitor Control Register*
			1	PMNCNTENSET	UNK	Count Enable Set Register, see the *ARM Architecture Reference Manual*
			2	PMNCNTENCLR	UNK	Count Enable Clear Register, see the *ARM Architecture Reference Manual*
			3	PMOVSR	UNK	Overflow Flag Status Register, see the *ARM Architecture Reference Manual*
			4	PMSWINC	UNK	Software Increment Register, see the *ARM Architecture Reference Manual*
			5	PMSELR	UNK	Event Counter Selection Register, see the *ARM Architecture Reference Manual*
			6	PMCEID0	0x3FFF0F3F	Common Event Identification Register 0, see the *ARM Architecture Reference Manual*
			7	PMCEID1	0x00000000	Common Event Identification Register 1, see the *ARM Architecture Reference Manual*
		c13	0	PMCCNTR	UNK	Cycle Count Register, see the *ARM Architecture Reference Manual*
			1	PMXEVTYPER	UNK	Event Type Selection Register, see the *ARM Architecture Reference Manual*
			2	PMXEVCNTR	UNK	Event Count Register, see the *ARM Architecture Reference Manual*
		c14	0	PMUSERENR	0x00000000	User Enable Register, see the *ARM Architecture Reference Manual*
c9	0	c14	1	PMINTENSET	UNK	Interrupt Enable Set Register, see the *ARM Architecture Reference Manual*
			2	PMINTENCLR	UNK	Interrupt Enable Clear Register, see the *ARM Architecture Reference Manual*
			3	PMOVSSET	UNK	Performance Monitor Overflow Flag Status Set Register, see the *ARM Architecture Reference Manual*
	1	c0	2	L2CTLR	0x00000000[a]	*L2 Control Register*
			3	L2ECTLR	0x00000000	*L2 Extended Control Register*

To access the register, specify CRn, CRm, and Op2 of each event. For more information, see the following: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0464f/BABFIBHD.html

The skeleton kernel module code, "pmuon.c" is provided on the class website in "pmuon" folder, and also copied below. It uses c9 registers to initialize the PMU.

[pmuon.c]

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

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your name");
MODULE_DESCRIPTION("PMUON");

int init_module(void) {
```

```
        unsigned int v;
        printk("Turn PMU on\n");

        // 1. Enable "User Enable Register"
        asm volatile("mcr p15, 0, %0, c9, c14, 0\n\t" :: "r" (0x00000001));

        // 2. Reset Performance Monitor Control Register(PMCR), Count Enable Set Register, and
Overflow Flag Status Register
        asm volatile ("mcr p15, 0, %0, c9, c12, 0\n\t" :: "r"(0x00000017));
        asm volatile ("mcr p15, 0, %0, c9, c12, 1\n\t" :: "r"(0x8000000f));
        asm volatile ("mcr p15, 0, %0, c9, c12, 3\n\t" :: "r"(0x8000000f));

        // 3. Disable Interrupt Enable Clear Register
        asm volatile("mcr p15, 0, %0, c9, c14, 2\n\t" :: "r" (~0));

        // 4. Read how many event counters exist
        asm volatile("mrc p15, 0, %0, c9, c12, 0\n\t" : "=r" (v)); // Read PMCR
        printk("pmon_init(): have %d configurable event counters.\n", (v >> 11) & 0x1f);


        // 5. Set event counter registers (Project Assignment you need to IMPLEMENT)

        return 0;
}

void cleanup_module(void) {
        printk("GOODBYE, PMU Off\n");
}
```

It accesses the c9 registers using inline assembly code (asm volatile). Two instruction types are used: "mcr" to write a register value, and "mrc" to read a register value. In each instruction, operands specify the target register. See the following link for additional information:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489g/Cihfifej.html

The skeleton code accomplishes the following:

1. Linux kernel does not allow a user-space program to access PMU without authorization. In order to access the PMU from the user-space program, set the "user enable bit" to '1' in the **User Enable Register** (PMUSERENR).
   Here is the "mcr" assembly line which writes a register value:

   ```
   asm volatile("mcr p15, 0, %0, c9, c14, 0\n\t" :: "r" (0x00000001));
   ```

   In the table, you can find that the {CRn, CRm, Op2} of PUSERNENR are {c9, c14, 0}.
   These three variables are the last three parameters of the "mcr" assembly instruction.
   "0x00000001" corresponds to the placeholder parameter "%0" in the asm volatile function call.
2. All counters can be initialized by configuring **Performance Monitor Control Register** (PMCR): http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0464f/BIIFDEEJ.html

Initialize the **Count Enable Set Register** (PMNCNTENSET) and **Overflow flag status register** (PMOVSR) in a similar way using their {CRn, CRm, Op2} values.

3. Disable the **Interrupt Enable Clear Register** (PMINTENCLR). The purpose of this register is to determine if any of the Performance Monitor Count Registers and the Cycle Count Register generate an interrupt on overflow. Any interrupt overflow enable bit written with a value of 1 clears the interrupt overflow enable bit to 0.
4. Read PMCR to know how many configurable event counters exist. The "mrc" assembly instruction is used for reading the register value into the "v" variable.

Cross-compile "pmuon.c" to generate the kernel module. Then, copy it to RPi2. Then, test by loading the kernel module.

$ su insmod pmuon.ko
$ dmesg | tail -2

```
[12643.359697] Turn PMU on
[12643.359740] pmuon_init(): have 4 configurable event counters
```

In the dmesg log you should see that CPU has 4 configurable event counters.

Assign events to performance counter registers in the kernel module

Modify the skeleton code ("pmuon.c") to assign the following events to four event counters:

- # of L1 data cache access
- # of L1 data cache miss
- # of L2 cache access
- # of L2 cache miss

We read from the four event counters one at a time, using the "Event Counter Selection Register" (PMSELR) and the "Event Type Selection Register" (PMXEVTYPER) in the kernel module.

1. **Event Counter Selection Register** (PMSELR)
    a. This register selects the event counter from which we will read.
    b. It should be set before setting an event type (PMXEVTYPER) or reading an event counter (PMXEVCNTR).
    c. You can select the event counter by setting one of the following bits:
    0x0 : First event counter
    0x1: Second event counter
    0x2: Third event counter
    0x3: Fourth event counter
2. **Event Type Selection Register** (PMXEVTYPER)
    a. After setting the PMSELR, specify the event type to count and read with event count registers. The following link shows all available events which Cortex-A7 can measure, and their event type IDs:
    http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0464f/BIIDBAFB.html
    b. After selecting it, access its actual counter value using PMXEVCNTR.

Submit your modified "pmuon.c" source code to select the four events for the event counters.

Read performance counter events in the user-space program

You will read the values of the four event counters during a given user-space program (the workload). Download the user-space program, "memmeasurement.c" and related files (governor_part1.c/.h and Makefile) from the class website ("memmeasurement" directory). Implement code which measures the cycle counter and the four performance counter events for "workload_body()" function.

The provided "main()" function performs the following tasks:

1. It initializes the following:
    a. "set_governor()" changes the CPU governor to "userspace". When the system uses the userspace governor, a user program can change the CPU frequency.
    b. "set_by_max_freq()" changes the CPU frequency to the maximum value. You may alternatively use "set_by_min_freq()". RPi2 supports two available CPU frequency settings. The two functions switch between the 900MHz and 600MHz frequency.
    c. "workload_init()" prepares an array to be accessed in "workload_body()" function. It also warms up the memory by accessing the array once.
2. Before running the "workload_body()" function, "reset_counter()" resets the cycle counter and the four event counters selected in the kernel module to zero by using PMCR. PMCR can be controlled by the user-space program since the kernel module enables the user enable register (PMUSERENR).
3. The code measures the execution time of "workload_body()" function using a system call, get_time_of_day().
4. After measuring the execution time, "workload_finish()" frees the array and change the governor from "userspace" to the original policy.

Once you install the part 1 kernel module, you can access the performance counter registers in any userspace program executed with root privilege (with sudo). For example, you can compile the original "memmeasurement" program as follows in your linux machine:

```
$ make CROSS_COMPILE=arm-linux-gnueabihf-
```

Copy the compiled kernel module and the user-space program binary into the RPi2. Then run the following:

```
$ sudo insmod pmuon.ko
$ sudo ./memmeasurement
Measurement start.
Exe_time: 3092172 us at 900000
$ sudo rmmod pmuon
```

Implement the code that measures the cycle counter and the four event counters (i.e., five counters in total) by modifying the provided "memmeasurement.c".

- Read the four event counters by reading "PMXEVCNTR". Don't forget to use "PMSELR" before accessing "PMXEVCNTR" to specify the event counter you want to read.
- Read the cycle counter by accessing PMCCNTR.

Submit your modified "memmeasurement.c" source code that measures the five performance counter registers for given "workload_body()" function.

Workload analysis based on performance counter measurement

You also need to write a report analyzing the workload (the "workload_body()" of memmeasurement.c) by using performance counters.

- Change the workload behavior by changing the three configuration variables:

```
static int DEF_ITERATION = 10000;
static int DEF_STRIDE = 128;
static int DEF_SIZE = 1024*1024;
```

  The value of DEF_ITERATION affects the execution time and the cycle count. The value of DEF_STRIDE affects the cache line. The value of DEF_SIZE changes the array size and affects the cache miss rates.
- Change the CPU frequency by using "set_by_max_freq()" and "set_by_min_freq()" .

Measure the performance counters of "workload_body()" over different configurations. Then, carefully explain your analysis result in the report. It should include answers to the following questions:

- How does the array size affect the cache miss rate? Why?
- What is the relationship between the miss rate and the sizes of L1 and L2 caches?
- How does the data size affect the execution time? Is it linear? Why or why not?
- How do the frequency and the cache misses affect the execution time? Why?

Plot the following results in your report over the min and max frequency of operation, and for array sizes ranging from 1K to 4MB:

  - execution time vs. array sizes
  - L1 and L2 cache miss number/rate vs. array sizes
  - execution time vs. cycle count
  - execution time vs. L1 and L2 cache miss numbers/rates
  - CPU energy vs. CPU frequency for different array sizes

Explain the results of the above plots that you will provide in your report.

Provide a plot, a table of the results, and a discussion of execution time and CPU energy vs. CPU freq. for array sizes of 8KB, 128KB, and 1MB. Use the table below to calculate the CPU energy values:

CPU Frequency	**Power**
900MHz	560 mW
600MHz	500 mW

Individual Project Submission Instructions (by 23:59:59 PST, Jan 21)

Submit the following via TED:

- Source code of your modified pmuon.c
    - It has to select the four event event counters:
      L1 cache access, L1 cache miss, L2 cache access, L2 cache miss
- Source code of your modified memmeasurement.c
    - It has to include the measurement code of the five performance counters: i.e.,
      cycle counter, L1 cache access, L1 cache miss, L2 cache access, L2 cache miss
- Report
    - Maximum 2pgs, 12pt Times New Roman font, excluding figures and table. The figures needed are specified above, and the table shows execution time and CPU energy vs. CPU freq. for array sizes of 8KB, 128KB, and 1MB.
    - Carefully explain what you learned by analyzing the workload over all data and frequency settings. Refer to the prompts in the report above as a guide.
    - Do not include your source code in the report.