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.