F28HS Hardware-Software Interface: Systems Programming

Hans-Wolfgang Loidl

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh

Semester 2 — 2025/26

[0]No proprietary software has been used in producing these slides

Outline

Tutorial 6: Performance Counters on the RPi 2

- Performance counters are hardware support for monitoring basic operations on the CPU
- They are very accurate and useful for monitoring resource consumption
- It is possible to count cycles, but also cache misses, (mispredicted) branches etc
- In this tutorial we will cover **how to use performance counters to get a precise measure of the runtime of a program**

HERIOT WATT UNIVERSITY

Architecture Support

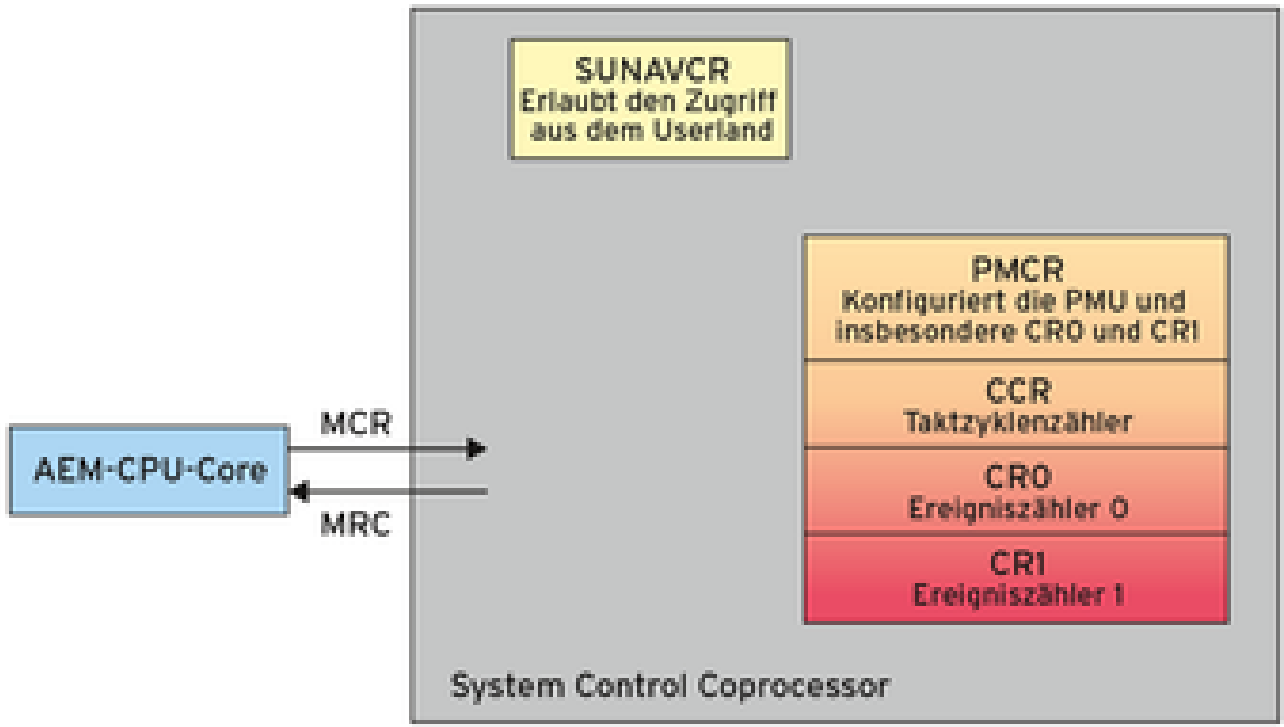
- BCM2835 (and later SoCs) provides a **Performance Monitoring Unit (PMU)** as a co-processor on the chip
- The unit supports in total 4 counter registers and a separate cycle counter register.
- These 4 registers can be configured to count a range of low-level events.
- There are 2 different interfaces for accessing this information.
  - the APB interface, which uses memory mapping and access registers on the PMU directly
  - the **CP15 interface**, which uses special assembler instructions for communicating between processor and PMU
- The PMU operations are usually not available for user programs (trying to run them directly will trigger an SIGILL exception)
- However, we can write a simple Linux kernel module to enable this functionality, and then use it through assembler instructions in our user code.

Overview: How to use the PMU

We need to go through the following steps:

1. Find out how to interact with the PMU
2. Enable access to the PMU from “user space”
3. Define what we want to monitor
4. Use access to the PMU to measure programs

Step 1: Find out how to interact with the PMU



The PMU is a **co-processor**, called **CP15**, separate from the main processor, but on the same chip.

The special assembler instructions `MRC` and `MCR` transfer data between processor register (`R`) and co-processor (`C`).

[0] From Linux Magazin 05/2015: Kerntechnik

Instructions for data transfer between processor and co-processor

- The ARM instruction set provides 2 instructions for the
  - `MCR`: Move to Coproc from ARM Reg
  - `MRC`: Move to ARM Reg from Coproc

The technical reference manual describes the instructions like this:

> To access the PMCR, read or write the CP15 registers with:
>
> ```
> MRC p15, 0, <Rt>, c9, c12, 0; Read Performance Monitor Control Register
> MCR p15, 0, <Rt>, c9, c12, 0; Write Performance Monitor Control Register
> ```

[0] See Cortex A7 MPcore Technical Reference Manual, Table 11-1 PMU register summary, p 241

Step 2: Enabling PMU access through a kernel module

- By default, the PMU can only be accessed in “privileged mode”, but this can be changed
- We need to construct a small Linux kernel module that enables the access to the PMU
- In essence, we need to embed some assembler instructions into an API pre-scribed by the Linux kernel
- For details on how to build a Linux kernel module see
  - The Linux Kernel Module Programming Guide, Peter Jay Salzman
  - Building instructions from a course on “Introduction to Embedded Computing” at Univ of California, San Diego, by Tajana Simunic Rosing
- Here, I’ll just shortly summarise the steps needed, and how to use performance monitoring in a simple example program

HERIOT WATT UNIVERSITY

Table 11-1: PMU registers

Table 11-1 PMU register summary

Register number	Offset	CRn	Op1	CRm	Op2	Name	Type	Description
0	0x000	c9	0	c13	2	PMXEVCNTR0	RW	Event Count Register, see the *ARM Architecture Reference Manual*
1	0x004	c9	0	c13	2	PMXEVCNTR1	RW	
2	0x008	c9	0	c13	2	PMXEVCNTR2	RW	
3	0x00C	c9	0	c13	2	PMXEVCNTR3	RW	
4-30	0x010-0x78	-	-	-	-	-	-	Reserved
31	0x07C	c9	0	c13	0	PMCCNTR	RW	Cycle Count Register, see the *ARM Architecture Reference Manual*
32-255	0x080-0x3FC	-	-	-	-	-		Reserved
256	0x400	c9	0	c13	1	PMXEVTYPER0	RW	Event Type Selection Register, see the *ARM Architecture Reference Manual*
257	0x404	c9	0	c13	1	PMXEVTYPER1	RW	
258	0x408	c9	0	c13	1	PMXEVTYPER2	RW	
259	0x40C	c9	0	c13	1	PMXEVTYPER3	RW	

[0] See Cortex A7 MPcore Technical Reference Manual, Table 11-1 PMU register summary, p 237

Table 11-1: PMU registers

897	0xE04	c9	0	c12	0	PMCR	RW	*Performance Monitor Control Register* on page 11-7
898	0xE08	c9	0	c14	0	PMUSERENR	RW	User Enable Register, see the *ARM Architecture Reference Manual*
899-903	0xE0C-0xE1C	-	-	-	-	-	-	Reserved

- The two main registers that we need to access are `PMCR` and `PMUSERENR`
  - `PMCR`: controls access to the PMU in general
  - `PMUSERENR`: is the **User Enable Register** that needs to be configured to allow user code to access the PMU

[0] See Cortex A7 MPcore Technical Reference Manual, Table 11-1 PMU register summary, p 237

Table 11-1: PMU registers

897	0xE04	c9	0	c12	0	PMCR	RW	*Performance Monitor Control Register* on page 11-7
898	0xE08	c9	0	c14	0	PMUSERENR	RW	User Enable Register, see the *ARM Architecture Reference Manual*
899-903	0xE0C-0xE1C	-	-	-	-	-	-	Reserved

- The two main registers that we need to access are `PMCR` and `PMUSERENR`
  - `PMCR`: controls access to the PMU in general
  - `PMUSERENR`: is the **User Enable Register** that needs to be configured to allow user code to access the PMU

[0] See Cortex A7 MPcore Technical Reference Manual, Table 11-1 PMU register summary, p 237

Structure of the PMCR register

To enable access to the PMU, we need to access the PMCR register. The **Performance Monitor Control Register (PMCR)** defines the core behaviour of the PMU:

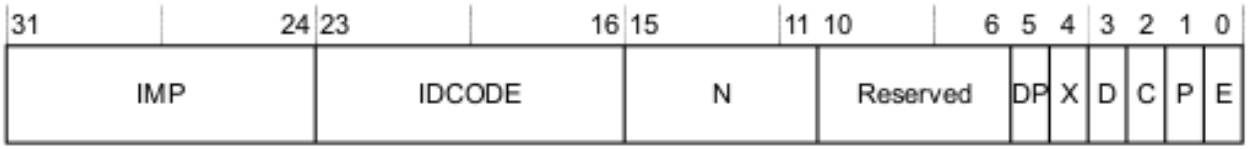


Figure 11-2 Performance Monitor Control Register bit assignments

[0] See Cortex A7 MPcore Technical Reference Manual, Figure 11-2 Performance Monitor Control Register bit assignments, p 240

The bits in the PMCR

Table 11-2 PMCR bit assignments (continued)

Bits	Name	Function
[4]	X	Export enable. This bit permits events to be exported to another debug device, such as a trace macrocell, over an event bus: **0** Export of events is disabled. This is the reset value. **1** Export of events is enabled. This bit is read/write.
[3]	D	Clock divider: **0** When enabled, PMCCNTR counts every clock cycle. This is the reset value. **1** When enabled, PMCCNTR counts once every 64 clock cycles. This bit is read/write.
[2]	C	Clock counter reset: **0** No action. This is the reset value. **1** Reset PMCCNTR to 0. This bit is write-only, and always RAZ.
[1]	P	Event counter reset: **0** No action. This is the reset value. **1** Reset all event counters, not including PMCCNTR, to 0. In Non-secure modes other than Hyp mode, writing a 1 to this bit does not reset event counters that the HDCR.HPMN field reserves for Hyp mode use. See *Hyp Debug Control Register* on page 4-68. In Secure state and Hyp mode, writing a 1 to this bit resets all event counters. This bit is write-only, and always RAZ.
[0]	E	Enable bit. Performance monitor overflow IRQs are only signaled when the enable bit is set to 1. **0** All counters, including PMCCNTR, are disabled. This is the reset value. **1** All counters are enabled. This bit is read/write.

Configuring the PMCR register

We are almost there!

The **encoding** for the PMCR register is (see Table 11-1): `c9, c12, 0`

We now configure the PMCR by setting the **E**, **P**, **C**, and **X** bits.
These are bits 0, 1, 2, and 4 in the PMCR register.
This means we need a bitmask of `0b00010111` or `0x17`.

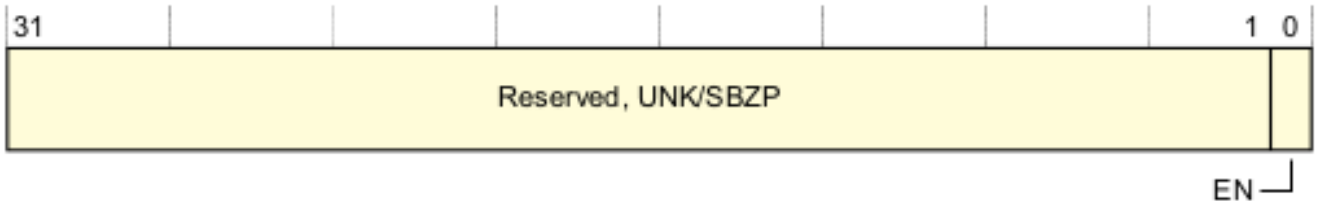
Here is the code:

```
mov r2, #0x17                  @ store bitmask 0x17 in reg r2
mcr p15, 0, r2, c9, c12, 0     @ transfer to PMCR
```

NB: For longer running programs you probably also want to enable the **D** bit, which divides the cylce counter by 64!

The PMUSERENR register

The PMUSERENR bit assignments are:



Bits[31:1] Reserved, UNK/SBZP.

EN, bit[0] User mode access enable bit. The possible values of this bit are:

- **0** User mode access to the Performance Monitors disabled.
- **1** User mode access to the Performance Monitors enabled.

Some MCR and MRC instruction accesses to the Performance Monitors are UNDEFINED in User mode when the EN bit is set to 0. For more information, see *Access permissions* on page C12-2330.

Accessing the PMUSERENR

To access the PMUSERENR, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c14, and <opc2> set to 0. For example:

```
MRC p15, 0, <Rt>, c9, c14, 0    : Read PMUSERENR into Rt
MCR p15, 0, <Rt>, c9, c14, 0    : Write Rt to PMUSERENR
```

[0] See ARM Architecture Reference Manual Cortex-A7, Sec B6.1.81, PMUSERENR, Performance Monitors User Enable Register, p 1924

Enabling access to the PMU

We can enable access to the PMU from "user space", from normal applications that are running outside the Linux "kernel space", by setting the lowest bit in the `PMUSERENR`:

```
mov r2, #0x01                  @ store bitmask 0x01 in reg r2
mcr p15, 0, r2, c9, c14, 0 @ transfer r2 to PMUSERENR
```

The `MCR` instruction transfers a value in a register to the co-processor. To find the **encoding** of the `PMUSERENR` we look up Table 11-1:
`c9, c14, 0`

[0] See also ARM Architecture Reference Manual Cortex-A7, Sec B5.8.2, Table B5-11: Summary of PMSA CP15 register descriptions, p 1796

Enabling access to the PMU

We also need to configure the following registers

- `PMCNTENSET`: Count Enable Set Register[1]:
  Purpose: The PMCNTENSET register enables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMNx. Reading this register shows which counters are enabled. This register is a Performance Monitors register.

- `PMOVSR`: Overflow Status Register `PMCNTENSET`: Count Enable Set Register[2]:
  Purpose: The PMOVSR holds the state of the overflow bits for:
  - *the Cycle Count Register, PMCCNTR*
  - *each of the implemented event counters, PMNx.*

  Software must write to this register to clear these bits.
  This register is a Performance Monitors register.

[1] See ARM Architecture Reference Manual Cortex-A7, Sec B6.1.74, p 1910
[2] See ARM Architecture Reference Manual Cortex-A7, Sec B6.1.78, p 1908

Table 11-1: PMCNTENSET and PMOVSR registers

We now have to find the register encodings for PMCNTENSET and PMOVSR.

Table 11-1 PMU register summary (continued)

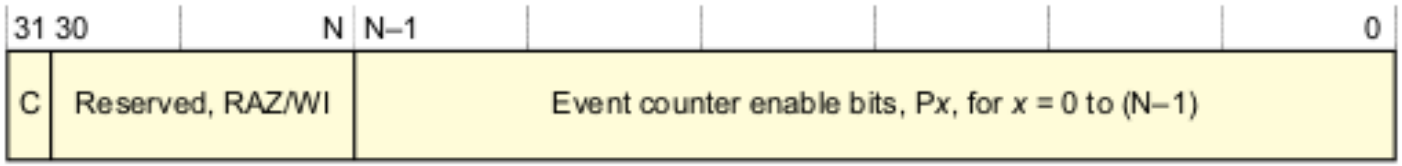
Register number	Offset	CRn	Op1	CRm	Op2	Name	Type	Description
800	0xC80	c9	0	c12	3	PMOVSR	RW	Overflow Flag Status Register, see the *ARM Architecture Reference Manual*
801-807	0xC84-0xC9C	-	-	-	-	-	-	Reserved
768	0xC00	c9	0	c12	1	PMCNTENSET	RW	Count Enable Set Register, see the *ARM Architecture Reference Manual*
769-775	0xC04-0xC1C	-	-	-	-	-	-	Reserved
776	0xC20	c9	0	c12	2	PMCNTENCLR	RW	Count Enable Clear Register, see the *ARM Architecture Reference Manual*
777-783	0xC24-0xC3C	-	-	-	-	-	-	Reserved

[2] See either Cortex A7 MPcore Technical Reference Manual, Figure 11-2 Performance Monitor Control Register bit assignments, p 240
or ARM Architecture Reference Manual Cortex-A7, Sec B5.8.2, Table B5-11: Summary of PMSA CP15 register descriptions, p 1796

Bits in PMCNTENSET and PMOVSR registers

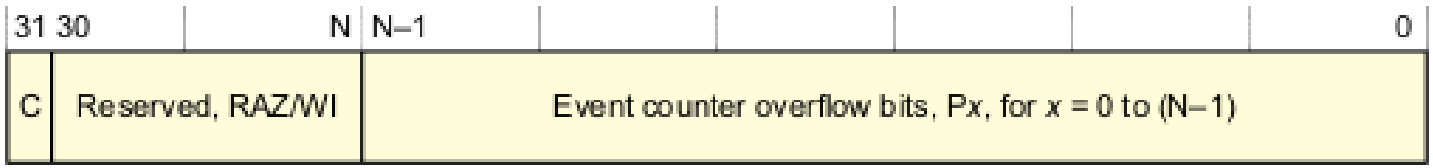
The PMCNTENSET register enables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMNx[3]

The PMCNTENSET register bit assignments are:



The PMOVSR holds the state of the overflow bit for: (i) the Cycle Count Register, PMCCNTR; (ii) each of the implemented event counters, PMNx.[4]

The PMOVSR bit assignments are:



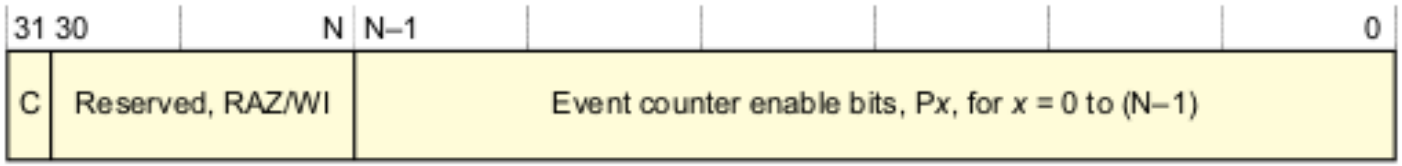
[3] See ARM Architecture Reference Manual Cortex-A7, Sec B4.1.116, p 1676

[4] See ARM Architecture Reference Manual Cortex-A7, Sec B4.1.116, p 1685

Bits in PMCNTENSET and PMOVSR registers

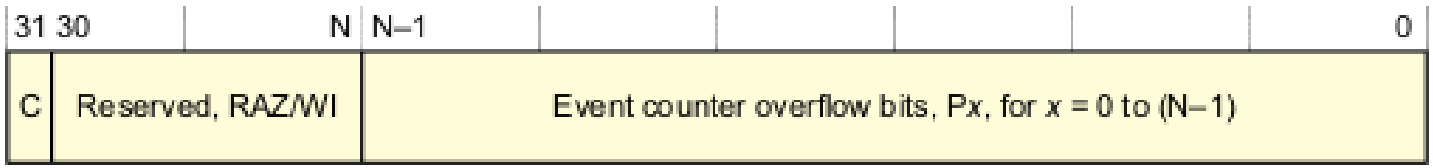
The PMCNTENSET register enables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMNx[3]

The PMCNTENSET register bit assignments are:



The PMOVSR holds the state of the overflow bit for: (i) the Cycle Count Register, PMCCNTR; (ii) each of the implemented event counters, PMNx.[4]

The PMOVSR bit assignments are:



[3] See ARM Architecture Reference Manual Cortex-A7, Sec B4.1.116, p 1676

[4] See ARM Architecture Reference Manual Cortex-A7, Sec B4.1.116, p 1685

Enabling access to the PMU

Almost there!

Both registers hold bitmasks over the event counters, to enable them and to control overflow.
We want to turn on the bit for every counter.
We have 4 counters in total, so we need to set the 4 least significant bits: we need a bitmask of `0b1111` or `0x0f`

Finally, here is the code to set the `PMCNTENSET` and `PMOVSR` registers:

```
mov r2, #0x0f                   @ store bitmask 0x0f in reg r2
mcr p15, 0, r2, c9, c12, 1      @ transfer to PMCNTENSET
mov r2, #0x0f                   @ store bitmask 0x0f in reg r2
mcr p15, 0, r2, c9, c12, 3      @ transfer to  PMOVSR
```

Step 3: Defining what to monitor

- Now that the PMU is enabled we need to decide what we want to monitor
- The PMU contains one cycle counter register, which we can use without special configuration: `PMCCNTR`
- The PMU contains 4 configurable counter registers
- For each of these registers we need to specify an **event type** to monitor

Table 16-1: PMU monitor events

Table 16-1 Performance monitor events

Number	Event counted
0x00	Software increment of the Software Increment Register
0x01	Instruction fetch that causes a Level 1 instruction cache refill
0x02	Instruction fetch that causes a Level 1 instruction TLB refill
0x03	Data Read or Write operation that causes a Level 1 instruction TLB refill
0x04	Data Read or Write operation that causes a Level 1 data cache access
0x05	Data Read or Write operation that causes a Level 1 data TLB refill
0x06	Memory-reading instruction executed
0x07	Memory-writing instruction executed
0x09	Exception taken
0x0A	Exception return executed
0x0B	Instruction that writes to the Context ID register
0x0C	Software change of program counter
0x0D	Immediate branch instruction executed
0x0F	Unaligned load or store
0x10	Branch mispredicted or not predicted
0x11	Cycle count; the register is incremented on every cycle

[4] From ARM Cortex-A Programmer's Guide, Table 16-1, p222

Table 16-1: PMU monitor events

0x11	Cycle count; the register is incremented on every cycle
0x12	Predictable branch speculatively executed
0x13	Data memory access
0x14	Level 1 instruction cache access
0x15	Level 1 data cache write-back
0x16	Level 1 data cache write-back
0x17	Level 2 data cache refill
0x18	Level 2 data cache write-back
0x19	Bus access
0x1A	Local memory error
0x1B	Instruction speculatively executed
0x1C	Instruction write to TTBR
0x1D	Bus cycle
0x1E-0x3F	Reserved

[4] From ARM Cortex-A Programmer's Guide, Table 16-1, p222

Defining what to monitor

We can define the events we want to monitor like this:

```
mov r2, #0x00                   @ counter #0
mcr p15, 0, r2, c9, c12, 5      @ transfer to PMSELR
mov r2, #0x11                   @ event type #11: cycle count
mcr p15, 0, r2, c9, c13, 1      @ transfer to PMXEVTYPER
```

The first 2 lines identify counter no. 0 (`0x00`) as the counter we are configuring.
The next 2 lines specify that this counter should monitor event no. `0x11`: instruction cycles.

Defining what to monitor

We can define the events we want to monitor like this:

```
mov r2, #0x00                   @ counter #0
mcr p15, 0, r2, c9, c12, 5      @ transfer to PMSELR
mov r2, #0x11                   @ event type #11: cycle count
mcr p15, 0, r2, c9, c13, 1      @ transfer to PMXEVTYPER
```

The first 2 lines identify counter no. 0 (`0x00`) as the counter we are configuring.
The next 2 lines specify that this counter should monitor event no. `0x11`: instruction cycles.

The complete kernel module

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

// 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"(0
    x00000017));
asm volatile ("mcr␣p15,␣0,␣%0,␣c9,␣c12,␣1\n\t" :: "r"(0
    x8000000f));
asm volatile ("mcr␣p15,␣0,␣%0,␣c9,␣c12,␣3\n\t" :: "r"(0
    x8000000f));

// 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);
```

Build the module

You first need to download the kernel sources.
To build the module, get the sample sources from `PMU_pmuon` and do this:

```
sudo make clean
sudo make
sudo insmod ./pmuon.ko
dmesg | tail
sudo rmmod pmuon
```

Step 4: Use the PMU in a user program

First we define macros for assembler 1-liners, which reset all counters (by writing to `PMCR`) and read the counters from the PMU:

```
#define armv7_reset_counters \
        asm volatile ("mcr␣p15,␣0,␣%0,␣c9,␣c12,␣0\n\t" :: "r"(0
            x00000017))  /* write to PMCR */

#define armv7_read_ccr( val ) \
        asm volatile("mrc␣␣␣p15,␣0,␣%0,␣c9,␣c13,␣0" : "=r"(val)
            )

#define armv7_read_cr0( val ) \
        asm volatile("mcr␣␣␣p15,␣0,␣%0,␣c9,␣c12,␣5" :: "r"(0x00
            )); /* select counter #0 */ \
        asm volatile("mrc␣␣␣p15,␣0,␣%0,␣c9,␣c13,␣2" : "=r"(val)
            )  /* read its value */
```

Measuring a simple C loop

The core of our user program is a counting loop:

```
armv7_reset_counters;
armv7_read_ccr( before_ccr );
armv7_read_cr0( before_cr0 );

for (i=0; i<n; i++ ) /* nothing */ ; // code to measure

armv7_read_ccr( after_ccr );
armv7_read_cr0( after_cr0 );
```

Example: running the measurement

```
> gcc -DCP15   -o rpi2-pmu01 rpi2-pmu01.c
> sudo ./rpi2-pmu01 10
Raspberry Pi 2 performance monitoring, using CP15 interface
The result is: 10
ccr: 338 (before: 0 after: 338) CYCLES
cr0: 338 (before: 6 after: 344) CYCLES
cr1: 12 (before: 0 after: 12) BRANCHES
cr2: 48 (before: 3 after: 51) CACHE HITS (Data read or write
    operation that causes a cache access at (at least) the
    lowest level of data or unified cache)
cr3: 32 (before: 0 after: 32) CACHE MISSES (Data read
    architecturally executed)
PMCR=41072011
Done.
```

Measuring assembler code

This is an assembler version of the counting loop:

```
armv7_reset_counters;
armv7_read_ccr( before_ccr );
armv7_read_cr0( before_cr0 );

asm volatile(/* inline assembler version of a counting loop */
 "_measure_me_asm_%=:\n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣MOVS␣␣␣␣␣R3,␣#0x00␣␣␣␣␣␣@␣initialise␣counter␣
    register\n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣B␣␣␣␣␣␣␣␣TEST%=␣@␣uncond.␣jump\n"
 "LOOP%=:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣@␣loop␣over␣counter␣R3\
    n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣ADD␣␣␣␣␣␣R3,␣R3,␣#1␣␣␣␣␣@␣increment␣counter
    ␣␣␣␣␣\n"
 "TEST%=:␣␣␣␣␣␣␣␣CMP␣␣␣␣␣␣R3,␣%[n]␣␣␣␣␣␣␣@␣test␣end␣value\n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣BLT␣␣␣␣␣␣LOOP%=\n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣␣␣MOV␣␣␣␣␣␣%[res],␣R3␣␣␣␣␣@␣done␣\n"
 : [res] "=r" (i) : [n] "r" (n) : "r3", "cc");

armv7_read_ccr( after_ccr );
armv7_read_cr0( after_cr0 );
```

Output

```
> gcc -DCP15 -o rpi2-pmu01 rpi2-pmu01.c
> sudo ./rpi2-pmu01 10
Raspberry Pi 2 performance monitoring, using CP15 interface
The result is: 10
ccr: 249 (before: 0 after: 249) CYCLES
cr0: 249 (before: 6 after: 255) CYCLES
cr1: 12 (before: 0 after: 12) BRANCHES
cr2: 7 (before: 3 after: 10) CACHE HITS (Data read or write
    operation that causes a cache access at (at least) the
    lowest level of data or unified cache)
cr3: 1 (before: 0 after: 1) CACHE MISSES (Data read
    architecturally executed)
PMCR=41072011
Done.
```

NB: we get precise runtime in machine-cycles; because we execute the loop 10 times (plus entry and exit), the branch counter shows 12; most operations work in registers, only a few memory access are needed and most of them can use the cache

```
armv7_reset_counters;
armv7_read_ccr( before_ccr );
armv7_read_cr0( before_cr0 );

asm volatile(/* inline assembler version of a counting loop
    with bad branch prediction */
 "_measure_me_asm_%=:\n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣MOVS␣␣␣␣␣R3,␣#0x00␣␣␣␣␣␣@␣initialise␣counter␣
    register\n"
 "TEST%=:␣␣␣␣␣␣␣CMP␣␣␣␣␣␣R3,␣%[n]␣␣␣␣␣␣␣@␣test␣end␣value\n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣BGE␣␣␣␣␣␣LEAVE%=␣␣␣␣␣␣␣@␣leave␣loop␣(BAD␣
    BRANCH␣PRED!)␣\n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣ADD␣␣␣␣␣␣R3,␣R3,␣#1␣␣␣␣␣@␣increment␣counter
    ␣␣␣␣␣\n"
 "\t␣␣␣␣␣␣␣␣␣␣␣␣␣␣B␣␣␣␣␣␣␣␣TEST%=␣␣␣␣␣␣␣␣␣@␣unconditional␣jump␣\
    n"
 "LEAVE%=:␣␣␣␣␣␣␣MOV␣␣␣␣␣␣%[res],␣R3␣␣␣␣␣@␣done␣\n"
 : [res] "=r" (i)  : [n] "r" (n) : "r3", "cc");

armv7_read_ccr( after_ccr );
armv7_read_cr0( after_cr0 );
```

Output

```
> gcc -DCP15 -o rpi2-pmu01 rpi2-pmu01.c
> sudo ./rpi2-pmu01 10
Raspberry Pi 2 performance monitoring, using CP15 interface
The result is: 10
ccr: 116 (before: 0 after: 116) CYCLES
cr0: 116 (before: 6 after: 122) CYCLES
cr1: 21 (before: 0 after: 21) BRANCHES
cr2: 7 (before: 3 after: 10) CACHE HITS (Data read or write
    operation that causes a cache access at (at least) the
    lowest level of data or unified cache)
cr3: 1 (before: 0 after: 1) CACHE MISSES (Data read
    architecturally executed)
PMCR=41072011
Done.
```

NB: In this case we have 21 rather than 12 branches, for the same kind of counting loop; this is because each iteration resulted in a mis-predicted branch, which was partially executed by the processor-pipeline, but then had to be aborted.

A larger user program: sum-and-average

Code example: `sumav3_asm_pmu.c`

Summary

- The ARM Cortex-A7 has an on-chip co-processor for hardware performance monitoring (PMU)
- The PMU can be configured to count a range of low-level events, e.g. cycles, branches, cache hits
- The PMU needs to be enabled from within a kernel module, so that user space programs can access it
- Once configured, inline assember instructions can be used to start/stop counting and read values
- The relevant assembler instructions are `MCR` and `MRC`, with a bespoke formatting of specifying registers on the CP15 co-processor (and on other on-chip co-processor)

HERIOT WATT UNIVERSITY