F28HS Hardware-Software Interface:
Systems Programming

Hans-Wolfgang Loidl

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

Semester 2 2016/17

No proprietary software has been used in producing these slides
Outline

1 Tutorial 1: Using Python and the Linux FS for GPIO Control
2 Tutorial 2: Programming an LED
3 Tutorial 3: Programming a Button input device
4 Tutorial 4: Inline Assembler with gcc
5 Tutorial 5: Programming an LCD Display
6 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.
Both the BCM2835 (of the RPi 1) and BCM2836 (of the RPi 2) provide 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)**.

0From *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 prescribed 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.
Table 11-1: PMU registers

<table>
<thead>
<tr>
<th>Register number</th>
<th>Offset</th>
<th>CRn</th>
<th>Op1</th>
<th>CRm</th>
<th>Op2</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x000</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>2</td>
<td>PMXEVCNTR0</td>
<td>RW</td>
<td>Event Count Register, see the ARM Architecture Reference Manual</td>
</tr>
<tr>
<td>1</td>
<td>0x004</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>2</td>
<td>PMXEVCNTR1</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0x008</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>2</td>
<td>PMXEVCNTR2</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x00C</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>2</td>
<td>PMXEVCNTR3</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>4-30</td>
<td>0x010-0x78</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>31</td>
<td>0x07C</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>0</td>
<td>PMCCNTR</td>
<td>RW</td>
<td>Cycle Count Register, see the ARM Architecture Reference Manual</td>
</tr>
<tr>
<td>32-255</td>
<td>0x080-0x3FC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>256</td>
<td>0x400</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>1</td>
<td>PMXEVTYPEPER0</td>
<td>RW</td>
<td>Event Type Selection Register, see the ARM Architecture Reference Manual</td>
</tr>
<tr>
<td>257</td>
<td>0x404</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>1</td>
<td>PMXEVTYPEPER1</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>258</td>
<td>0x408</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>1</td>
<td>PMXEVTYPEPER2</td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>259</td>
<td>0x40C</td>
<td>c9</td>
<td>0</td>
<td>c13</td>
<td>1</td>
<td>PMXEVTYPEPER3</td>
<td>RW</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>897</td>
<td>0xE04</td>
<td>PMCR</td>
</tr>
<tr>
<td>898</td>
<td>0xE08</td>
<td>PMUSERENR</td>
</tr>
<tr>
<td>899-903</td>
<td>0xE0C-0xE1C</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

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.

---

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

---

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](image)

---

^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)**

<table>
<thead>
<tr>
<th>Bits</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
</table>
| [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:

```assembly
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 cycle counter by 64!
The **PMUSERENR** register

The PMUSERENR bit assignments are:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved, UNK/SBZP</td>
</tr>
<tr>
<td>1-0</td>
<td>User mode access enable bit</td>
</tr>
</tbody>
</table>

**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

---

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

---

0See 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, PMN$x$. 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, PMN$x$.

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

---

\(^2\) 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*
The **PMCNTENSET** register enables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMN\(x^3\).

---

### The PMCNTENSET register bit assignments are:

<table>
<thead>
<tr>
<th>31 30</th>
<th>N  N-1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Reserved, RAZ/WI</td>
<td>Event counter enable bits, P(x), for (x = 0) to ((N-1))</td>
</tr>
</tbody>
</table>

---

The PMOVSR holds the state of the overflow bit for: (i) the Cycle Count Register, PMCCNTR; (ii) each of the implemented event counters, PMN\(x^4\).

---

### The PMOVSR bit assignments are:

<table>
<thead>
<tr>
<th>31 30</th>
<th>N  N-1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Reserved, RAZ/WI</td>
<td>Event counter overflow bits, P(x), for (x = 0) to ((N-1))</td>
</tr>
</tbody>
</table>

---


Bits in \textbf{PMCNTENSET} and \textbf{PMOVSR} registers

The \textbf{PMCNTENSET} register enables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMN\textsubscript{x}³

The \textbf{PMCNTENSET} register bit assignments are:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>N</td>
<td>N-1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>Reserved, RAZ/WI</td>
<td></td>
<td></td>
<td>Event counter enable bits, P\textsubscript{x}, for x = 0 to (N-1)</td>
<td></td>
</tr>
</tbody>
</table>

The \textbf{PMOVSR} holds the state of the overflow bit for: (i) the Cycle Count Register, PMCCNTR; (ii) each of the implemented event counters, PMN\textsubscript{x}.⁴

The \textbf{PMOVSR} bit assignments are:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>N</td>
<td>N-1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>Reserved, RAZ/WI</td>
<td></td>
<td></td>
<td>Event counter overflow bits, P\textsubscript{x}, for x = 0 to (N-1)</td>
<td></td>
</tr>
</tbody>
</table>

³See ARM Architecture Reference Manual Cortex-A7, Sec B4.1.116, p 1676
⁴See 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 \texttt{0b1111} or \texttt{0x0f}

Finally, here is the code to set the \texttt{PMCNTENSET} and \texttt{PMOVSR} registers:

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

---

4 From ARM Cortex-A Programmer’s Guide, Table 16-1, p222
### Table 16-1: PMU monitor events

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>Cycle count; the register is incremented on every cycle</td>
</tr>
<tr>
<td>0x12</td>
<td>Predictable branch speculatively executed</td>
</tr>
<tr>
<td>0x13</td>
<td>Data memory access</td>
</tr>
<tr>
<td>0x14</td>
<td>Level 1 instruction cache access</td>
</tr>
<tr>
<td>0x15</td>
<td>Level 1 data cache write-back</td>
</tr>
<tr>
<td>0x16</td>
<td>Level 1 data cache write-back</td>
</tr>
<tr>
<td>0x17</td>
<td>Level 2 data cache refill</td>
</tr>
<tr>
<td>0x18</td>
<td>Level 2 data cache write-back</td>
</tr>
<tr>
<td>0x19</td>
<td>Bus access</td>
</tr>
<tr>
<td>0x1A</td>
<td>Local memory error</td>
</tr>
<tr>
<td>0x1B</td>
<td>Instruction speculatively executed</td>
</tr>
<tr>
<td>0x1C</td>
<td>Instruction write to TTBR</td>
</tr>
<tr>
<td>0x1D</td>
<td>Bus cycle</td>
</tr>
<tr>
<td>0x1E-0x3F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Defining what to monitor

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

```assembly
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:

```assembly
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,c0,%0,c9,c14,0
	" :: "r" (0
  x00000001));

// 2. Reset Performance Monitor Control Register (PMCR), Count Enable Set Register, and Overflow Flag Status Register
asm volatile ("mcr_p15,c0,%0,c9,c12,0
	" :: "r"(0
  x00000017));
asm volatile ("mcr_p15,c0,%0,c9,c12,1
	" :: "r"(0
  x8000000f));
asm volatile ("mcr_p15,c0,%0,c9,c12,3
	" :: "r"(0
  x8000000f));

// 3. Disable Interrupt Enable Clear Register
asm volatile("mcr_p15,c0,%0,c9,c14,2
	" :: "r" (~0));

// 4. Read how many event counters exist
asm volatile("mrc_p15,c0,%0,c9,c12,0
	" : "=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:

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

```c
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:

```asm
asm volatile(/* inline assembler version of a counting loop */
  "_measure_me_asm_%=:
  "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 );
> gcc -DCP15 -o rpi2-pmu01 rpi2-pmu01.c
> sudo ./rpi2-pmu01 10

Raspberry Pi 2 performance monitoring, using CP15 interface
The result is: 10

cr: 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)

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_%=:
"\t_________MOVSR3,#0x00@ initialise_counter_register
"TEST%=:\t_________CMPR3,%[n]@ test_end_value
"\t_________BGEREAVER=,leave_loop (BAD_BRANCH_PRED!)
"\t_________ADD,R3,R3,#1@ increment_counter
"\t_________BTEST%=,unconditional_jump
"LEAVE%=:\t_________MOV%[res],R3@ done
: [res] "=r" (i) : [n] "r" (n) : "r3", "cc");

armv7_read_ccr( after_ccr );
armv7_read_cr0( after_cr0 );
> 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

ccr0: 116 (before: 6 after: 122) CYCLES

ccr1: 21 (before: 0 after: 21) BRANCHES

ccr2: 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)

ccr3: 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 assembler instructions can be used to start/stop counting and read values.
- The relevant assembler instructions are \texttt{MCR} and \texttt{MRC}, with a bespoke formatting of specifying registers on the CP15 co-processor (and on other on-chip co-processors).