Valtrix Blog

Programming ARM v8 Performance Monitors

(/programming/programming-armv8-performance-counters)

May 25, 2016. | By: Shubhodeep Roy Choudhury

Verification engineers often need a measure for the quality of the test stimulus being generated. *Is the instruction sequence generated by a test configuration for cache eviction really meeting its intent? How do we find if no coverage is being generated after a recent source commit in the test generator tool? How does your test program fare against the industry benchmarks or another program?* The answer to all these questions lies in programming and analysis of performance monitoring unit (PMU) counters. PMU in processor implementations play a very important role in design verification. In addition to providing a low-overhead access to a large number of counters/events related to functional units in CPU, it forms a basis to conduct low level performance analysis and tuning.

ARM based systems typically implement an optional non-invasive performance monitoring debug component based on Performance Monitor Unit (PMU) architecture. The ARMv8 applicatons include version 3 of the performance monitors extension i.e. PMUv3. These performance monitors are fairly accurate in the measurements of the different architectural and microarchitectural events supported by the specification.

In this blog, I will explain the programming sequence (along with the pseudo code) for the ARM PMUv3 performance monitors. This is based on the recent implementation in Sting (http://valtrix.in/sting/) and can be re-used for any other program.

Overview of Performance Monitors

I am going to list the high level features of performance monitors extension (PMUv3). The implementation provides a 64-bit cycle counter to give an approximation of time as measred by the performance counters. The specification also provides 32 event counters, the event for which is programmable. Although, ARMv8 provides register space for upto 31 conters, the actual number is implementation defined. For example, the number of event counters implemented in Qualcomm Dragonboard 410C is only 6. The cycle counter is always implemented, even if the other event counters are not.

The specification provides control for enabling and resetting the counters, flagging overflows, enabling interrupts on overflows etc. The PMU architectures uses event numbers to identify different events which can be broadly categorized into architectural, microarchitectural and implementation specific. The actual events which are enabled is again implementation defined.

Please refer the ARMv8 specification to get the complete list of events supported by PMU. In the next section, I will provide a brief overview of the registers which users need to program in order to collect the performance monitoring data.

Performance Monitors Registers

The performance monitors registers provide a handy programming interface to collect the coverage numbers. I am only going to need a small subset for collecting the data. A brief description of those registers is provided below -

1. **PMCEID0/PMCEID1** (Common event identification register 0 and 1)
   - This register defines the events the common architectural and microarchitectural events that are implemented.
   - If a particular bit is set to 1, then that particular event is implemented.
2. **PMCR** (Control Register)
   - This register provides details of the performance monitoring implementation.
   - The number of event counters implemented (PMCR.N) can be found by reading bits[15:11] of this register.
   - It can be used to configure and control the event counters.
3. **PMCNTENSET** (Count enable set register)
   - This register is used to enable the cycle count register and any other event counters implemented.
   - The bitmask for the event counters which need to be enabled needs to be programmed in this register.
4. **PMCCFILTR** (Cycle count filter register)
   - This register determines the modes in which the cycle counter operates.
   - It is useful if the performance monitoring code intended to be used is run in different exception levels.
5. **PMCCNTR** (Cycle count register)
   - This register hold the value of processor cycle counter.
   - It can be programmed to increment every cycle or every 64th cycle.
6. **PMSELR** (Event counter selection register)
   - This register can be used to select the current event counter or the cycle counter.

- A value of 0 to PMCR.N - 1 can be programmed in PMSELR.SEL to select one of the implemented event counters.
- A value of 31 can be programmed to select the cycle counter.
- This enables us to access all the event counters in a programmatic manner.

7. **PMXEVCNTR** (Selected event count register)
   - This register can be used to read or write the value of the selected event counter programmed in PMSELR.SEL.
8. **PMXEVTYPER** (Selected event type register)
   - This register can be used to access the event type register for the selected event counter programmed in PMSELR.SEL.

These registers can be read or written using MRS and MSR instructions respectively. I will give the pseudo code for programming the registers and collecting the performance monitoring data.

Pseudo Code for Register Programming

I would like to list down all the caveats before I proceed to the pseudo code.

1. The use case given below only measures the performance monitoring data for a finite length code section. As a result, we do not expect any of the counters to overflow. Hence, I am not providing any polling/interrupt based mechanism to handle the same at this point.
2. I am using two API calls - read_register() and write_register(), to access the performance monitors register. The user can implement it their environment appropriately.
3. Assuming another API call - is_evnt_supported() to be present, which takes the event number as input and reports whether it is supported or not after checking the common event identification registers - PMCEID0/1.
4. The events enabled and counted are limited by the number of event counters. In case N event counters are implemented, only the first N enabled events are profiled. The code can be easily enhanced to profile a list of events specified by the user. I will leave that enhancement to the user.

A setup code first needs to be run in order to record few implementation specific parameters which will later be used to start and stop counting.

```
//
// Pseudo code for setup
//

pmcr_val       = read_register(PMCR)
num_evnt_cntrs = (pmcr_val >> 11) & 0x1F

if (num_evnt_cntrs == 0)
{
    print "Only cycle counter implemented"
}
else
{
    pmceid0_val = read_register(PMCEID0)
    pmceid1_val = read_register(PMCEID1)
}

// Enable user mode access of performance monitoring
// registers in case the user wants to access it from
// EL0
pmuserenr_val = read_register(PMUSERENR)
write_register(PMUSERENR, pmuserenr_val | 0xD)
```

This piece of code needs to be run before the start of the instruction stream which needs to be profiled for the performance monitoring data.

```
//
// Pseudo code for starting the counting
//

i = 0
for (j = 0; j < 64; j++)
{
    if (i >= num_evnt_cntrs)
        break

    if (is_evnt_supported(j) == true)
    {
        write_register(PMSELR, i++)
        write_register(PMXEVTYPER, 0x8000000 | j)
    }
}

write_register(PMCCFILTR, 0x8000000)

// Set the bitmask in PMCNTENSET to enable cycle
// counter and all the implemented event counters
write_register(PMCNTENSET, 0x80000000 | ((0x1 << num_evnt_cntrs) - 1))

// Start counting by setting PMCR.E after clearing all the
// counters by setting PMCR.P and PMCR.C
pmcr_val = read_register(PMCR)
write_register(pmcr_val | 0x7)
```

This piece of code needs to be run just after the end of the instruction stream. This will stop the counting and the counters can be read and recorded afterwards.

```
//
// Pseudo code for stopping the counting
//

// Stop counting by resetting PMCR.E and writing the
// bitmask in PMCNTENCLR to disable cycle counter and
// all the implemented event counters
write_register(PMCNTENCLR, 0x80000000 | ((0x1 << num_evnt_cntrs) - 1))
pmcr_val = read_register(PMCR)
write_register(PMCR, pmcr_val & ~0x7)

pmue_ccntr = read_register(PMCCNTR)
print "CPU Cycle Counter: %x", pmue_ccntr

i = 0
for (j = 0; j < 64; j++)
{
    if (i >= num_evnt_cntrs)
        break

    if (is_evnt_supported(j) == true)
    {
        write_register(PMSELR, i++)
        pmue_evcntr = read_register(PMXEVCNTR)
        print "PMU Event Number %d Counter: %x", j, pmue_evcntr
    }
}
```

Conclusion

Performance monitoring is a wonderful feature which allows us to take a sneak peek into the low level operations of the hardware. I hope the readers will find the information useful and can easily use it in their environment. Feel free to contact me in case there are any questions.

Tweet Share G+ 分享

Categories: Programming (/Category/#)

Subscribe

Subscribe to this blog via RSS (/feed.xml).

Categories

General (/category/#general)	2
Programming (/category/#programming)	4
Announcements (/category/#announcements)	3
Reports (/category/#reports)	1

Recent Posts

Valtrix at the 8th RISC-V Workshop Barcelona May 2018 (/announcements/valtrix-riscv-8th-workshop)
Posted on 17 Apr 2018

Running STING on PULPino Platform (/programming/running-sting-on-pulpino)
Posted on 18 Dec 2017

Test Plans for RISC-V CPU Specification (/announcements/riscv-test-plan)
Posted on 10 Nov 2017

Support for Accellera's Upcoming Portable Stimulus Specification in STING (/general/portable-stimulus-wg)
Posted on 02 Oct 2016

Popular Categories

General (2) (/category/#general) Programming (4) (/category/#programming)

Announcements (3) (/category/#announcements) Reports (1) (/category/#reports)