Coresight CPU Debug Module

Author: Leo Yan <leo.yan@linaro.org>
Date: April 5th, 2017

Introduction

Coresight CPU debug module is defined in ARMv8-a architecture reference manual (ARM DDI 0487A.k) Chapter 'Part H: External debug', the CPU can integrate debug module and it is mainly used for two modes: self-hosted debug and external debug. Usually the external debug mode is well known as the external debugger connects with SoC from JTAG port; on the other hand the program can explore debugging method which rely on self-hosted debug mode, this document is to focus on this part.

The debug module provides sample-based profiling extension, which can be used to sample CPU program counter, secure state and exception level, etc; usually every CPU has one dedicated debug module to be connected. Based on self-hosted debug mechanism, Linux kernel can access these related registers from mmio region when the kernel panic happens. The callback notifier for kernel panic will dump related registers for every CPU; finally this is good for assistant analysis for panic.

Implementation

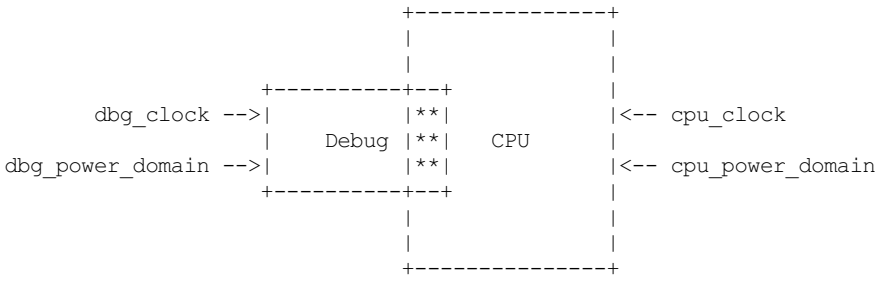
- During driver registration, it uses EDDEVID and EDDEVID1 - two device ID registers to decide if sample-based profiling is implemented or not. On some platforms this hardware feature is fully or partially implemented; and if this feature is not supported then registration will fail.
- At the time this documentation was written, the debug driver mainly relies on information gathered by the kernel panic callback notifier from three sampling registers: EDPCSR, EDVIDSR and EDCIDSR: from EDPCSR we can get program counter; EDVIDSR has information for secure state, exception level, bit width, etc; EDCIDSR is context ID value which contains the sampled value of CONTEXTIDR_EL1.
- The driver supports a CPU running in either AArch64 or AArch32 mode. The registers naming convention is a bit different between them, AArch64 uses 'ED' for register prefix (ARM DDI 0487A.k, chapter H9.1) and AArch32 uses 'DBG' as prefix (ARM DDI 0487A.k, chapter G5.1). The driver is unified to use AArch64 naming convention.
- ARMv8-a (ARM DDI 0487A.k) and ARMv7-a (ARM DDI 0406C.b) have different register bits definition. So the driver consolidates two difference:

  If PCSROffset=0b0000, on ARMv8-a the feature of EDPCSR is not implemented; but ARMv7-a defines "PCSR samples are offset by a value that depends on the instruction set state". For ARMv7-a, the driver checks furthermore if CPU runs with ARM or thumb instruction set and calibrate PCSR value, the detailed description for offset is in ARMv7-a ARM (ARM DDI 0406C.b) chapter C11.11.34 "DBGPCSR, Program Counter Sampling Register".

  If PCSROffset=0b0010, ARMv8-a defines "EDPCSR implemented, and samples have no offset applied and do not sample the instruction set state in AArch32 state". So on ARMv8 if EDDEVID1.PCSROffset is 0b0010 and the CPU operates in AArch32 state, EDPCSR is not sampled; when the CPU operates in AArch64 state EDPCSR is sampled and no offset are applied.

Clock and power domain

Before accessing debug registers, we should ensure the clock and power domain have been enabled properly. In ARMv8-a ARM (ARM DDI 0487A.k) chapter 'H9.1 Debug registers', the debug registers are spread into two domains: the debug domain and the CPU domain.



For debug domain, the user uses DT binding "clocks" and "power-domains" to specify the corresponding clock source and power supply for the debug logic. The driver calls the pm_runtime_{put|get} operations as needed to handle the debug power domain.

For CPU domain, the different SoC designs have different power management schemes and finally this heavily impacts external debug module. So we can divide into below cases:

- On systems with a sane power controller which can behave correctly with respect to CPU power domain, the CPU power

domain can be controlled by register EDPRCR in driver. The driver firstly writes bit EDPRCR.COREPURQ to power up the CPU, and then writes bit EDPRCR.CORENPDRQ for emulation of CPU power down. As result, this can ensure the CPU power domain is powered on properly during the period when access debug related registers;

- Some designs will power down an entire cluster if all CPUs on the cluster are powered down - including the parts of the debug registers that should remain powered in the debug power domain. The bits in EDPRCR are not respected in these cases, so these designs do not support debug over power down in the way that the CoreSight / Debug designers anticipated. This means that even checking EDPRSR has the potential to cause a bus hang if the target register is unpowered.

  In this case, accessing to the debug registers while they are not powered is a recipe for disaster; so we need preventing CPU low power states at boot time or when user enable module at the run time. Please see chapter "How to use the module" for detailed usage info for this.

Device Tree Bindings

See Documentation/devicetree/bindings/arm/coresight-cpu-debug.txt for details.

How to use the module

If you want to enable debugging functionality at boot time, you can add "coresight_cpu_debug.enable=1" to the kernel command line parameter.

The driver also can work as module, so can enable the debugging when insmod module:

```
# insmod coresight_cpu_debug.ko debug=1
```

When boot time or insmod module you have not enabled the debugging, the driver uses the debugfs file system to provide a knob to dynamically enable or disable debugging:

To enable it, write a '1' into /sys/kernel/debug/coresight_cpu_debug/enable:

```
# echo 1 > /sys/kernel/debug/coresight_cpu_debug/enable
```

To disable it, write a '0' into /sys/kernel/debug/coresight_cpu_debug/enable:

```
# echo 0 > /sys/kernel/debug/coresight_cpu_debug/enable
```

As explained in chapter "Clock and power domain", if you are working on one platform which has idle states to power off debug logic and the power controller cannot work well for the request from EDPRCR, then you should firstly constraint CPU idle states before enable CPU debugging feature; so can ensure the accessing to debug logic.

If you want to limit idle states at boot time, you can use "nohlt" or "cpuidle.off=1" in the kernel command line.

At the runtime you can disable idle states with below methods:

It is possible to disable CPU idle states by way of the PM QoS subsystem, more specifically by using the "/dev/cpu_dma_latency" interface (see Documentation/power/pm_qos_interface.rst for more details). As specified in the PM QoS documentation the requested parameter will stay in effect until the file descriptor is released. For example:

```
# exec 3<> /dev/cpu_dma_latency; echo 0 >&3
...
Do some work...
...
# exec 3<>-
```

The same can also be done from an application program.

Disable specific CPU's specific idle state from cpuidle sysfs (see Documentation/admin-guide/pm/cpuidle.rst):

```
# echo 1 > /sys/devices/system/cpu/cpu$cpu/cpuidle/state$state/disable
```

Output format

Here is an example of the debugging output format:

```
ARM external debug module:
coresight-cpu-debug 850000.debug: CPU[0]:
coresight-cpu-debug 850000.debug:  EDPRSR:  00000001 (Power:On DLK:Unlock)
coresight-cpu-debug 850000.debug:  EDPCSR:  handle_IPI+0x174/0x1d8
coresight-cpu-debug 850000.debug:  EDCIDSR: 00000000
coresight-cpu-debug 850000.debug:  EDVIDSR: 90000000 (State:Non-secure Mode:EL1/0 Width:64bits VMID:0)
coresight-cpu-debug 852000.debug: CPU[1]:
coresight-cpu-debug 852000.debug:  EDPRSR:  00000001 (Power:On DLK:Unlock)
coresight-cpu-debug 852000.debug:  EDPCSR:  debug_notifier_call+0x23c/0x358
coresight-cpu-debug 852000.debug:  EDCIDSR: 00000000
coresight-cpu-debug 852000.debug:  EDVIDSR: 90000000 (State:Non-secure Mode:EL1/0 Width:64bits VMID:0)
```