**CPU内核使用内存时取消数据缓存**

唐松泉

2023/8/26

**历史记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **责任人** | **时间** | **备注** |
| V1.0 | 唐松泉 | 2023/8/25 | 初版 |
|  |  |  |  |

目录

[1 问题背景 3](#_Toc144110277)

[2 问题分析 3](#_Toc144110278)

[3 问题解决思路 3](#_Toc144110279)

[3.1 方案可行性分析 3](#_Toc144110280)

[3.1.1 禁用dcache 3](#_Toc144110281)

[3.1.2 使用dma 4](#_Toc144110282)

[3.1.3 及时清dcache 4](#_Toc144110283)

[3.1.4 禁止某块DDR内存区域使用dcache 5](#_Toc144110284)

[4 方案实现 5](#_Toc144110285)

[4.1 平台配置 5](#_Toc144110286)

[4.2 寄存器配置 6](#_Toc144110287)

[4.2 确定访问寄存器的异常等级 7](#_Toc144110288)

[4.3 代码实现 10](#_Toc144110289)

[5 其他注意事项 11](#_Toc144110290)

[参考文档 12](#_Toc144110291)

# 1 问题背景

xi-linx Zynq UltraScale+平台，PL端和PS共享了一个块内存块，PS写入数据，PL读出数据，PS这端的数据是来自上位机通过网络接收的数据，从DDR写入读出的数据一致；但PL端从内存读出来的数据和PS写进去的不一样，调试发现PS这端重复写两次PL才能到读到和PS写进的数据。

# 2 问题分析

根据调试现象，且PS这端使用CPU 对应core访问ddr时，使用了缓存dcache，而PL端访问时没有用到缓存，定位该问题是PS这端有缓存一致性问题，即cpu为提高读写效率，将写入的数据仍留在dcache中，dcache满时才会被刷到ddr中。

# 3 问题解决思路

解决缓存一致性问题，目前大多数的解决方法：

1. 禁用dcache
2. 使用dma，不经过CPU，这样就没有缓存一致性问题
3. 及时清dcache，将写入的数据清到DDR
4. 禁止某块DDR内存区域使用dcache

# 3.1 方案可行性分析

### 3.1.1 禁用dcache

禁用dcache虽然可解决缓存一致性问题，但这将大大降低CPU访问内存效率，仅为了一个功能牺牲所有性能，不值得，该方案不可行。

### 3.1.2 使用dma

使用dma是目前解决缓存一致性问题最常用的方法：即PS这端的两块内存块的数据拷贝动作，都使用dma来做，避开了CPU访问使用缓存的问题，xi-linx Zynq UltraScale+平台集成了几个AXI dma IP内存块的搬运，根据手册，PS和PL之间，PS作为主设备的通路有HPM0\_FPD、HPM1\_FPD、HPM0\_LPD, 可以用于PS控制PL设备；PL作为主设备的通路有HP0、HP1、HP2、HP3、HPC0、HPC1、ACP、ACE, 可以用于PL访问PS的DDR，使PS和PL通过共享内存交互数据。

理论上可行，但从上位机发送到PL端的数据，仍然需要CPU来处理写入DDR，即CPU接收到的数据，由于是CPU本身进行DDR写入，一样会用到缓存，而当前方案DMA，并不支持DMA从缓存搬数据。

根据以上信息，该方案也不可行。

### 3.1.3 及时清dcache

清dcache是出现该问题时直接能想到的，也是可行的，ARM也提供了对应的寄存器清空dcache，下图是从armv8寄存器手册查到的，CISW指令支持用户清空icache缓存，使得当前缓存数据被清空后备注为无效：

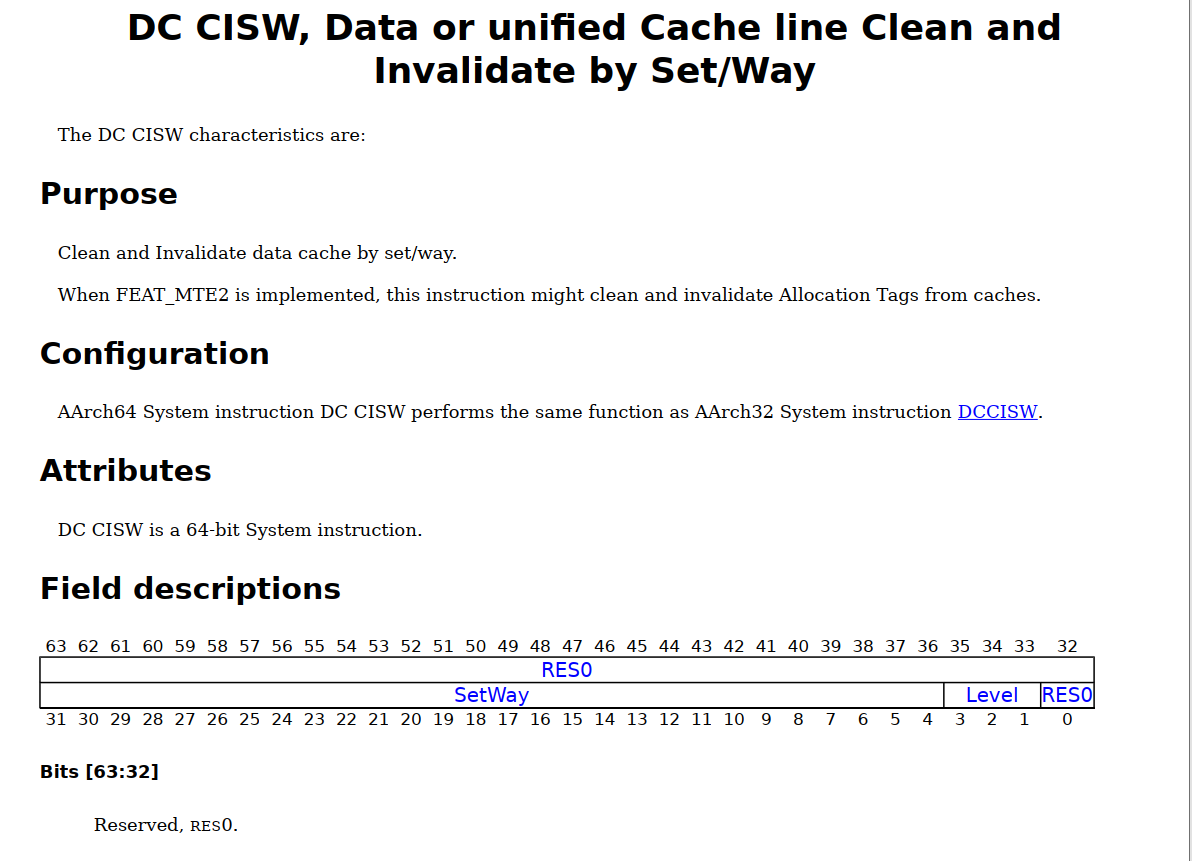


fig1 armv8寄存器 CISW

但清空缓存有个效率的问题，每次拷贝之后都要清空一次缓存，比直接关闭对应内存使用dcache效率来的低些。

### 3.1.4 禁止某块DDR内存区域使用dcache

该方法是目前能想到比较周全的方案，仅仅禁止PL和PS共享内存的内存块，让PL写一次，PS即可以马上拿到对应的数据，Arm也提供了对应的寄存器来配置：

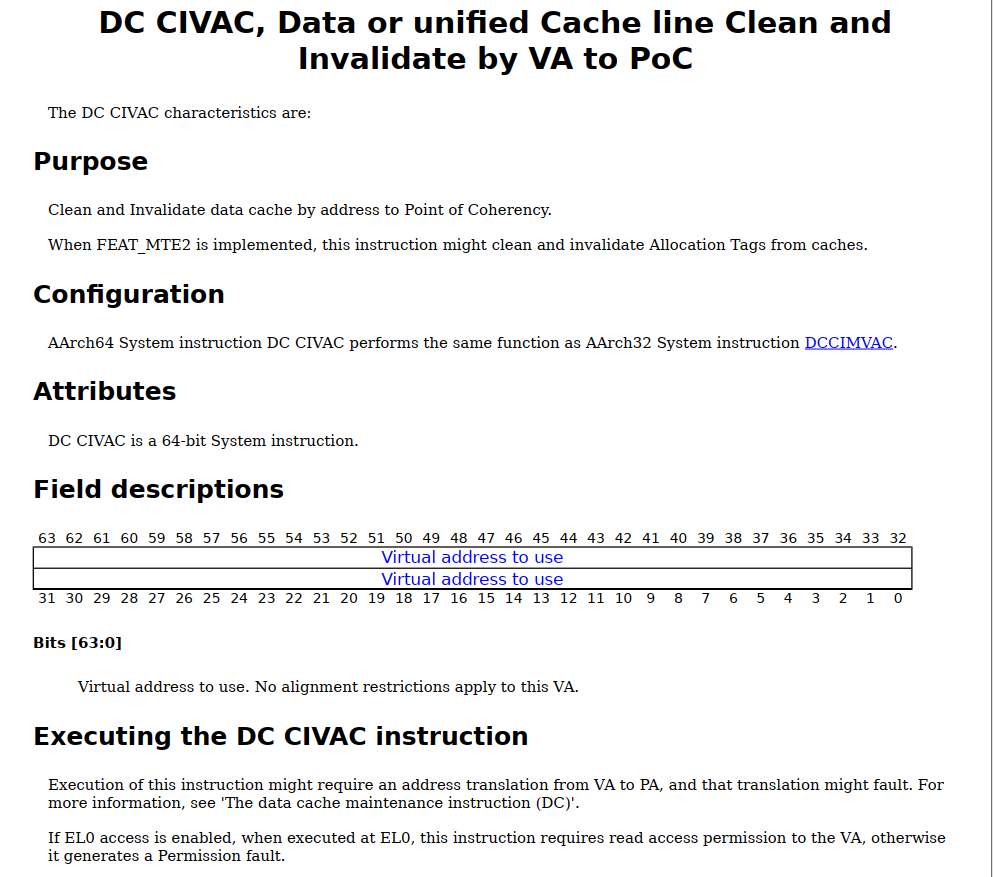


fig3 armv8 寄存器CIVAC

# 4 方案实现

## 4.1 平台配置

需要先明确当前平台的配置，从芯片手册上查到，xi-linx Zynq UltraScale+平台集成了4个cortex-a53的core，每个core都带有32K的icache和dcache，CPU内核支持aarch64和aarch32结构，从工程编译使用上看，当前正在用的全是aarch64架构（ARCH=arm64，包括工具链使用的都是aarch64），所以后续所有的寄存器，都是用armv8 aarch64架构的寄存器。

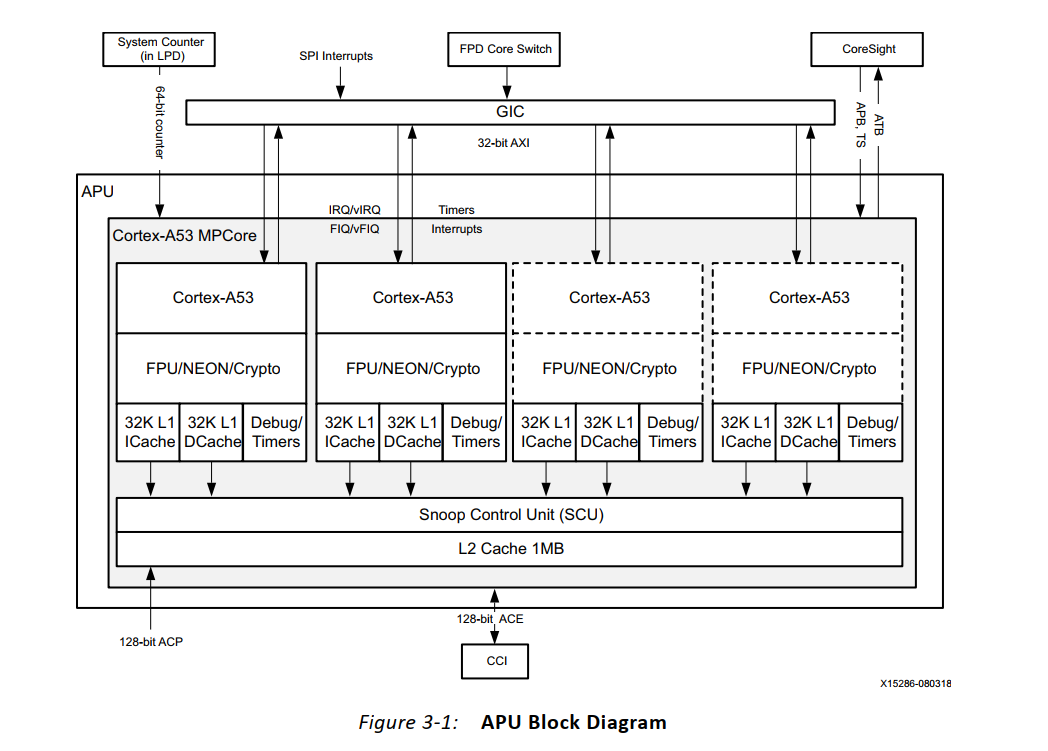
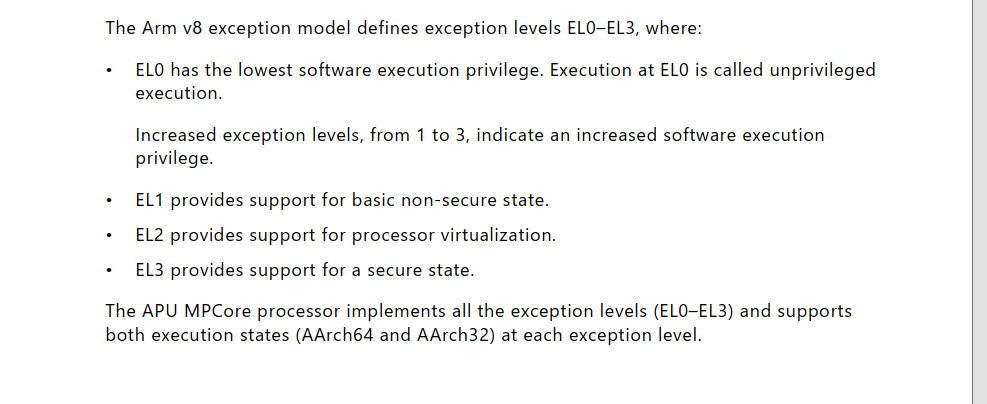


fig3 Zynq UltraScale+ CPU cores

另外，该平台的CPU几个内核，硬件上都实现了EL0-EL3四个异常等级：，且EL0/EL1只有non-secure状态，这对后面分析寄存器的使用特别重要。



## 4.2 寄存器配置

从arm官方手册armv8-registers可以查到DC CIVAC命令用来配置地址到PoC的缓存行无效化，即对应地址将不再使用dcache，CIVAC寄存器的描述如下：

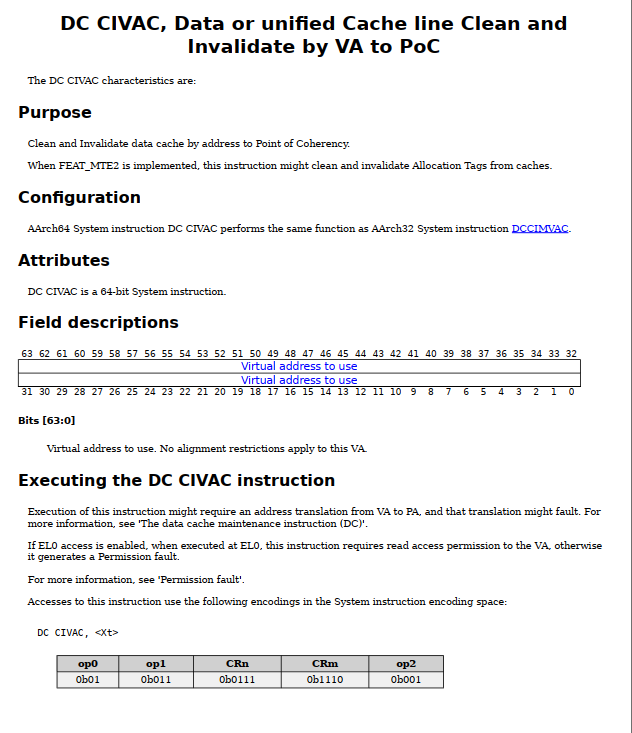


fig4 CIVAC

比较重要的信息：操作的地址必须是虚拟地址，操作地址需要行对齐，一行(line)的大小是64字节。

## 4.2 确定访问寄存器的异常等级

因为跑的是Linux，默认情况下，进入Linux内核状态，core处于EL1异常等级，在应用层则处于EL0，这决定了，要配置这个寄存器，需要在EL1或者EL0异常等级访问。

ARMV8提供了一个寄存器：HCR\_EL2

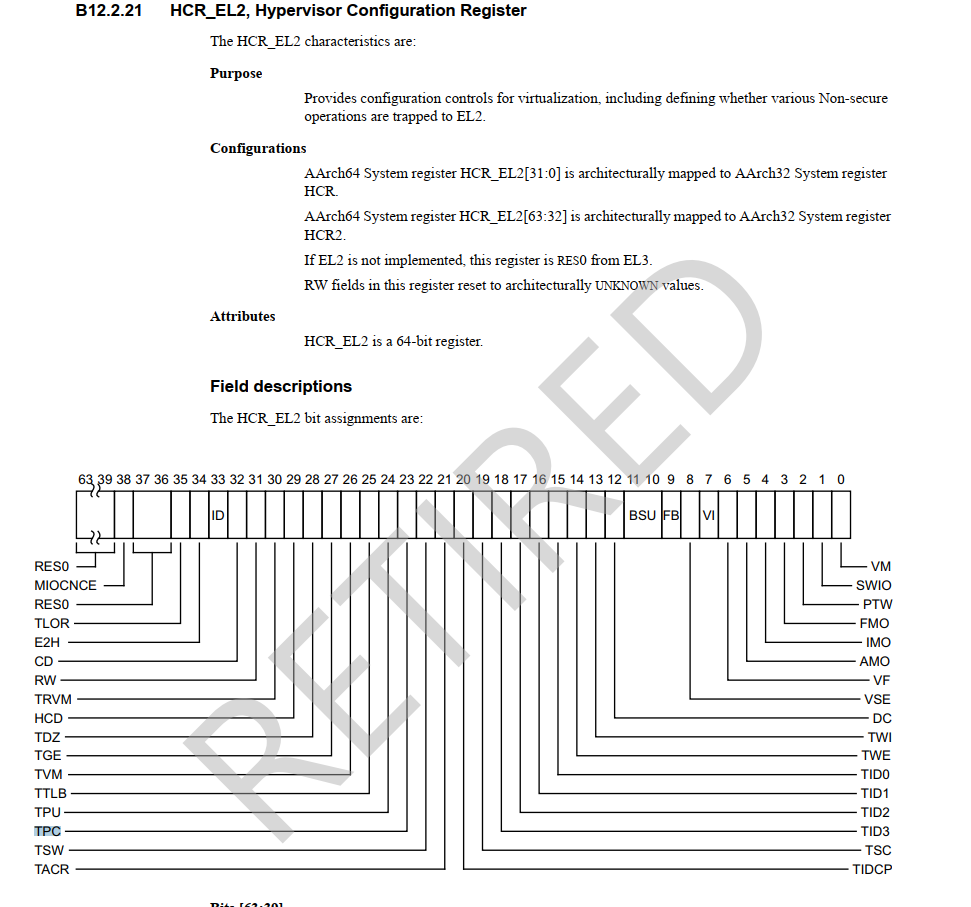


fig5 HCR\_EL2寄存器

该寄存器的bit[23] 是对EL0/1数据或统一缓存维护(PoC/Persistence)指令的捕获，当该bit[23]配置为0时，系统不会在用户使用non-secure-EL1/EL0状态下访问寄存器时在EL2异常等级捕获异常：

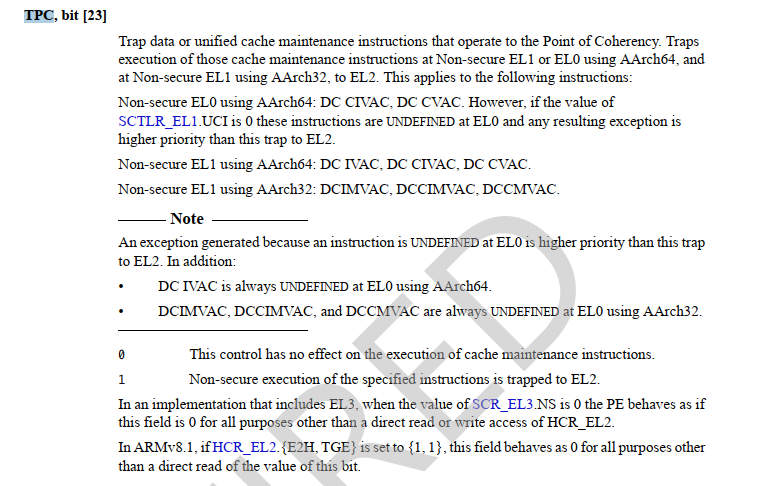


fig6 HCR\_EL2 bit[23]描述

而且，在non-secure EL1异常等级下，配置该寄存器的指令是DC CIVAC（aarch64）。

从内核代码看，只有在启动脚本配置了HCR\_EL2这个寄存器：

………

SYM\_INNER\_LABEL(init\_el2, SYM\_L\_LOCAL)

    mov\_q   x0, HCR\_HOST\_NVHE\_FLAGS

    msr hcr\_el2, x0

    isb

    init\_el2\_state

    /\* Hypervisor stub \*/

    adr\_l   x0, \_\_hyp\_stub\_vectors

    msr vbar\_el2, x0

    isb

    /\*

     \* Fruity CPUs seem to have HCR\_EL2.E2H set to RES1,

     \* making it impossible to start in nVHE mode. Is that

     \* compliant with the architecture? Absolutely not!

     \*/

    mrs x0, hcr\_el2

    and x0, x0, #HCR\_E2H

cbz x0, 1f

………

该寄存器初始化被配置成HCR\_HOST\_NVHE\_FLAGS这个宏的值，该宏值定义在sdk\build\tmp\work-shared\zynqmp-generic\kernel-source\arch\arm64\include\asm\kvm\_arm.h

……

#define HCR\_VIRT\_EXCP\_MASK (HCR\_VSE | HCR\_VI | HCR\_VF)

#define HCR\_HOST\_NVHE\_FLAGS (HCR\_RW | HCR\_API | HCR\_APK | HCR\_ATA)

#define HCR\_HOST\_NVHE\_PROTECTED\_FLAGS (HCR\_HOST\_NVHE\_FLAGS | HCR\_TSC)

#define HCR\_HOST\_VHE\_FLAGS (HCR\_RW | HCR\_TGE | HCR\_E2H)

/\* TCR\_EL2 Registers bits \*/  
………

该值并没有配置bit[23]

即bit[23]的值为默认值0，可知CIVAC这个寄存器是可以在异常等级non-scure-EL1/EL0访问且不会被EL2捕获异常，但EL0的访问，还需要另外一个寄存器的配置：SCTLR\_EL1

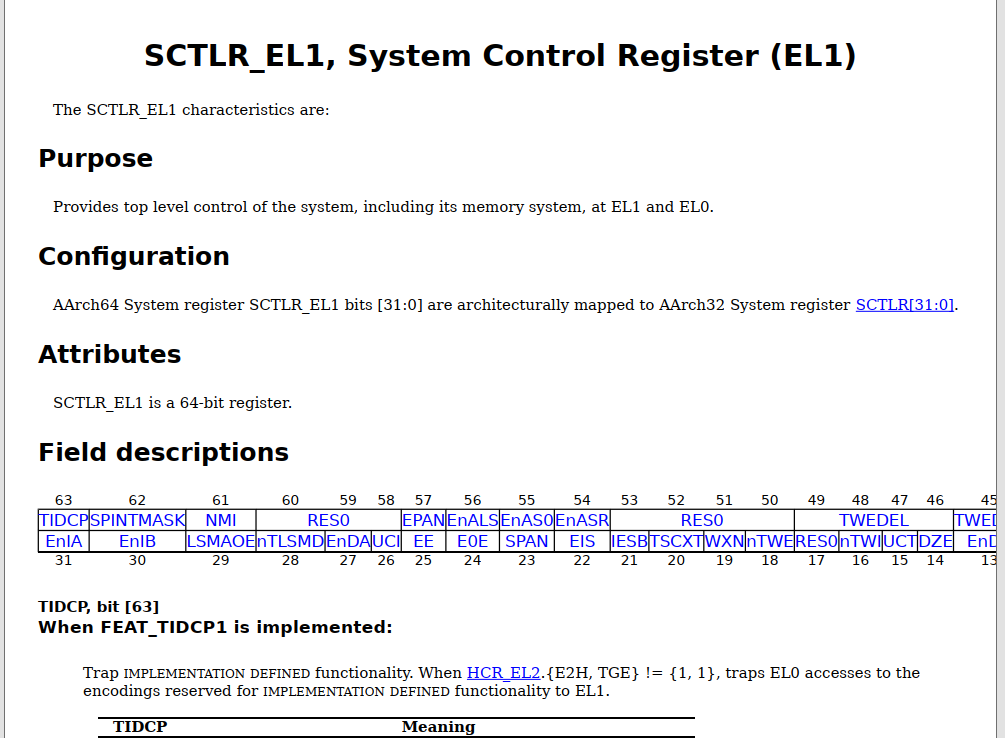


fig7 SCTLR\_EL1 寄存器

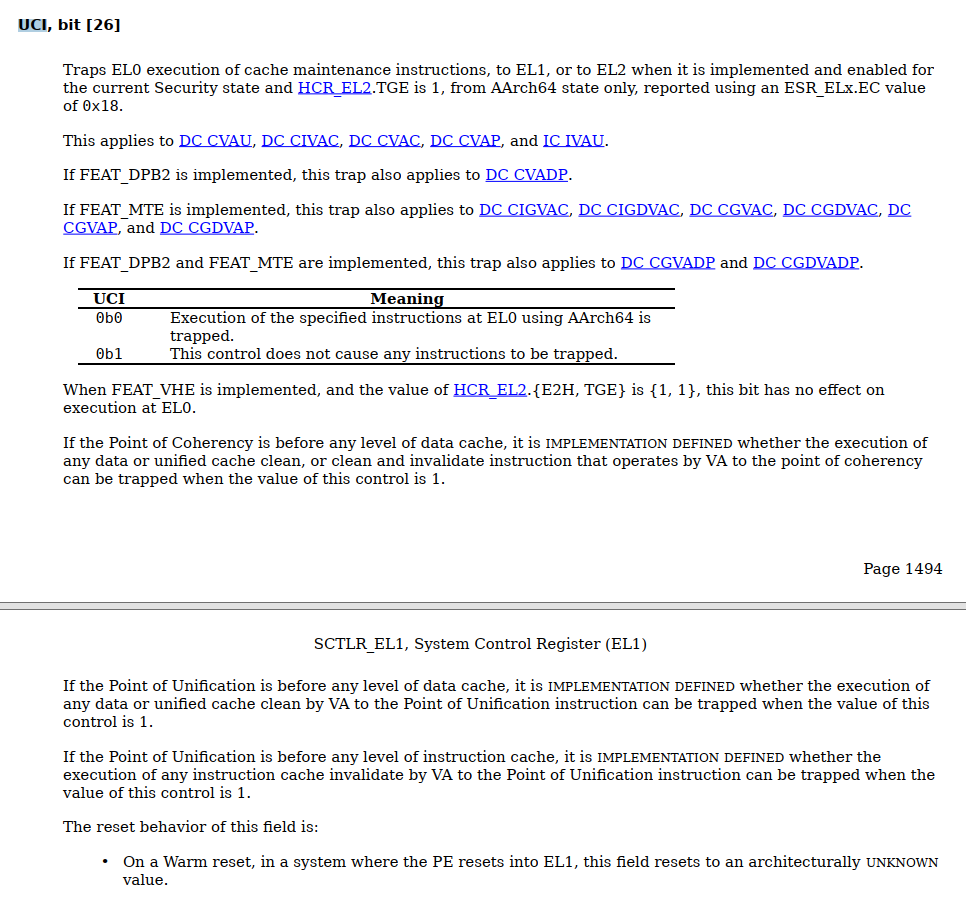


fig8 SCTLR\_EL1 寄存器bit[26]

该寄存器bit[26]用来开启EL0访问CIVAC这个寄存器，该bit为1时，EL0下可以访问DC CVAU, DC CIVAC, DC CVAC, DC CVAP, and IC IVAU 这些寄存器，否则将产生异常。

在Linux内核中，SCTLR\_EL1的配置也是在启动脚本：

SYM\_INNER\_LABEL(init\_el1, SYM\_L\_LOCAL)

    mov\_q   x0, INIT\_SCTLR\_EL1\_MMU\_OFF

msr sctlr\_el1, x0

………

msr sctlr\_el1, x20          // disable the MMU

………

脚本对该寄存器的配置，也只是对MMU的相关操作，并没有对bit[26]配置，bit[26]保持了默认值0，即默认下CIVAC这个寄存器不能在EL0下访问，否则内核将抛出异常。

由以上分析，在Linux系统中，最终能访问CIVAC的异常等级只能在EL1，除非去配置对应寄存器sctlr\_el1，但该寄存器也需要在el1的等级下配置，实际上也是只能在EL1下去访问。

以上分析，最终决定了只能在内核做内存dcache的配置，在不对当前内核做任何改动下，只能用驱动的方式来做这个配置。

## 4.3 代码实现

#define mtcpdc(reg,val) \_\_asm\_\_ \_\_volatile\_\_("dc " #reg ",%x0" : : "r" (val))

/\* Data Synchronization Barrier \*/

#define dsb() \_\_asm volatile ("dmb sy" ::: "memory")

static int set\_memory\_invalidate\_dcache(unsigned long paddr, unsigned long len)

{

    const unsigned long cacheline = 64U;

    unsigned int \*vaddr = phys\_to\_virt(paddr);

    unsigned long adr = (unsigned long)vaddr;

    unsigned int end = adr + len;

    adr = adr & (~0x3F);

    local\_irq\_disable();

    if (len != 0U) {

        while (adr < end) {

            mtcpdc(CIVAC,adr);

            adr += cacheline;

#if defined (VERSAL\_NET)

            \_\_asm volatile( "NOP" );

………

#endif

        }

    }

    /\* Wait for invalidate to complete \*/

    dsb();

    local\_irq\_enable();

    return 0;

}

需要注意的是，dsb等待完成的操作，用aarch64架构工具链编译，不能写成

\_\_asm volatile ("dsb" ::: "memory")

需要写成

\_\_asm volatile ("dmb sy" ::: "memory")

否则编译阶段将产生如下错误

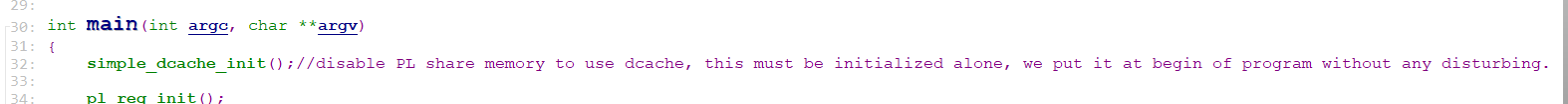
missing immediate expression at operand 1 -- 'dmb'

网上的说法是，这个是由于cpsr寄存器，在arm64被移除导致的。

参考：https://github.com/rsta2/uspi/issues/19

# 5 其他注意事项

该配置，个人认为需要单独运行，不能被其他线程干扰，如果其他线程正在使用缓存，特别是使用配置的内存地址范围，此时又去配置该内存，可能会产生意想不到的异常，最好时在程序运行最开始时初始化。



# 参考文档

《SysReg\_xml\_v88A-2021-12.pdf》

《DDI0557A\_b\_armv8\_1\_supplement\_RETIRED》

《DDI0500J\_cortex\_a53\_trm.pdf》

《ug1085-zynq-ultrascale-trm.pdf》