**ARM与Cortex笔记-ARM MPCore (Multi-Processor Core) 多核心架构解析.**

ARM与Cortex笔记–ARM MPCore (Multi-Processor Core) 多核心架构解析.

<http://loda.hala01.com/2011/06/arm%E8%88%87cortex%E7%AD%86%E8%A8%98-arm-mpcore-multi-processor-core-%E6%9E%B6%E6%A7%8B%E8%A7%A3%E6%9E%90/>

hlchou@mail2000.com.tw

by loda

随着目前Smart Phone的应用与复杂度增加,这类消费性电子产品,必须要能在考虑功耗与持久性的前提下,达成使用者可接受的高效能,并且还要有足够的使用与待机时间,基于如此,ARM MPCore多核心架构,就是一个在消费性电子产品上可以考虑的处理器架构选择.

参考ARM网站有关Cortex A9 Performance的介绍([**http://www.arm.com/products/processors/cortex-a/cortex-a9.php**](http://www.arm.com/products/processors/cortex-a/cortex-a9.php) ),以双核心架构在TSMC 40G性能优化的版本中,效能与功耗比为5.26 (DMIPS/mW),而Cortex A9单一核心的Dhrystone 性能比为2.50 DMIPS/MHz,也就是说,如果希望要达到10000DMIPS,以单核心而言就需要达到2GHz的频率,此时的总功耗约为1.9W (= 10000/5.26),但如果是以多核心的架构,就可以在频率不需要大幅拉到2GHz的情况下,透过增加处理器数量,来达到所期待的运算DMIPS总数,而且系统功耗增加幅度也低(当然芯片的面积会增加),可以参考这份ARM的投影片“Parallel Computing in your Pocket” (参考网址:http://www.iet-cambridge.org.uk/arc/seminar07/slides/JohnGoodacre.pdf),可以看到单核心要达到三核心的效能时,透过拉高频率达到一样的效能时,核心的总功耗会是采用三核心方案的三倍以上,也就是说,如果操作系统上有适度的Muti-Task多任务,用多核心去满足总体效能的期待,会比单核心的架构,更符合省电的功耗效益.

在多核心的架构下,还需要去检视操作系统的排程机制,若OS Scheduling Tick是透过Interrupt Distributor发给每个处理器,则每个处理器都会透过执行排程的程序代码,选择需要执行的Task,进行Task Context-Switch的流程,这样的想法对于让系统Best-Effort运作是比较合理的,但对于消费性产品而言,若可以让系统尽可能省电,而又只会减损些微的效能,反而是种加分,

也因此,除了要确保每个处理器可以透过WFI进入省电状态外,若是把OS Scheduling Tick固定发给Primary Processor,其它目前没有Task执行的处理器,就进入WFI的状态,由该Primary Processor进行排程与工作分配,non-Primary Processor接收到来自Primary Processor的IPI中断后,就会被唤醒,进行对应的Task Context-Switch动作,如果系统是处于不忙碌的状态下,就会有机会让non-Primary Processor的处理器,有机会维持比较长时间的StandBy休眠的状态,MPCore的消费性产品,可以维持比较长时间的运作.

本文主要以ARMv7 与CortexA9为主要讨论的范围(可供参考的文件例如:ARMv7-AR Architecture Reference Manual),所讨论的内容,主要以笔者认为值得深入讨论的项目,并不一定符合每个人对于ARM MPCore所需的范围,最后,本文虽尽可能提供正确的信息,若有不尽完善之处,请以ARM的文件为依归.

**多核心架构的概念**

多核心的架构下,开机时,只会有一个处理器在运作称为Primary Processor(或导引处理器BSP “bootstrap processor “),其它处理器则称为 non-Primary Processor(或应用处理器AP “Application processor”),在系统初始化与关机过程中,都是由Primary Processor来负责,主要执行如下流程

1,Invalidate Data Cache

2,Invalidate SCU(Snoop Control Unit) duplicate tags for all processors

3,Invalidate L2 Cache

4,Enable SCU

5,Enable Data Cache

6. Enable L2 Cache

7. Set SMP mode with ACTLR.SMP.

等到操作系统初始一个段落后,才会去启动其它的non-Primary Processor,并由这些处理器执行如下流程

1. Invalidate Data Cache

2. Enable Data Cache

3. Set SMP with ACTLR.SMP.

整个系统的排程机制就会根据这些启动的Processor来分派工作,如果有用不到的处理器资源,也可以透过WFI(Wait For Interrupr)让处理器处于省电的模式.

基于多处理器的架构,处理器也会提供硬件层级支持内存同步的指令,例如:LDREX, STREX, SWP与 SWPB,避免当有一个以上的处理器对同一个内存内容存取时,当该内容在处理器A中有做变动,以致使该内容在处理器B中失效时,可以透过这类指令确保处理器B可以同步到最新的内容,维持整个系统运作的正确性.

而在操作系统中SpinLock的机制,也会透过LDREX/STREX来进行,确保当操作系统进行SpinLock动作时,可以得到硬件层级的内存同步确保,避免SpinLock在多核心架构下的运作失误. (透过软件,要做到比硬件支持更有效率的自旋锁是相对困难),

参考Linux Kernel在ARM平台上的SpinLock实作(档案位置arch/arm/include/asm/spinlock.h),如下所示

static inline void \_\_raw\_spin\_lock(raw\_spinlock\_t \*lock)

{

unsigned long tmp;

\_\_asm\_\_ \_\_volatile\_\_(

“1: **ldrex** %0, [%1]\n”

” teq %0, #0\n”

#ifdef CONFIG\_CPU\_32v6K

” wfene\n”

#endif

” **strexeq** %0, %2, [%1]\n”

” teqeq %0, #0\n”

” bne 1b”

: “=&r” (tmp)

: “r” (&lock->lock), “r” (1)

: “cc”);

smp\_mb();

}

static inline int \_\_raw\_spin\_trylock(raw\_spinlock\_t \*lock)

{

unsigned long tmp;

\_\_asm\_\_ \_\_volatile\_\_(

” **ldrex** %0, [%1]\n”

” teq %0, #0\n”

” **strexeq** %0, %2, [%1]”

: “=&r” (tmp)

: “r” (&lock->lock), “r” (1)

: “cc”);

if (tmp == 0) {

smp\_mb();

return 1;

} else {

return 0;

}

}

此外,

1, 处理器之间可透过IPI(Inter-Processor Interrupt)彼此沟通,

2,处理器之间是合作关系,彼此没有从属的关系,

3,所有的处理器都看到同样的内存空间,彼此所寻址的物理内存空间也是一样,在同样的内存位置上都是存取同样的内存内容.并且,共同基于同一个操作系统程序代码,来对所有的处理器进行Task的排程工作.

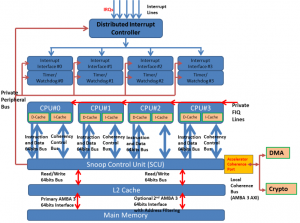
4,每个处理器都是以Task为单位去进行多任务,但进入到Kernel Mode (Ring0 or Supervisor Mode)时,所看到的内存内容是同一块被保护的区间,但是切到User Mode (Ring 3 or User Mode)时,就是根据MMU看到各自的内存区块.

5,所有的处理器都共享同样的I/O周边与中断控制器,每个处理器都可以收到来自任何周边的中断触发.

**检视ARM MPCore架构**

ARM系列在ARM11(例如:ARM1176)时,就已经导入Multi-Processor Core的架构,Cortex系列,包括A5,A9跟A15都支持MPCore的架构,目前可以支持4个核心架构(可以参考文件: ARM11 MPCore Processor Technical Reference Manual in [**http://infocenter.arm.com/help/topic/com.arm.doc.ddi0360f/DDI0360F\_arm11\_mpcore\_r2p0\_trm.pdf**](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0360f/DDI0360F_arm11_mpcore_r2p0_trm.pdf)或 Cortex-A9 MPCore Technical Reference Manual in[**http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407e/DDI0407E\_cortex\_a9\_mpcore\_r2p0\_trm.pdf**](http://infocenter.arm.com/help/topic/com.arm.doc.ddi0407e/DDI0407E_cortex_a9_mpcore_r2p0_trm.pdf) ), 多核心的架构会透过Snoop Control Unit接口同步每个处理器各自的L1 Data Cache内容,并以Distributed Interrupt Controller支持既有的ARM Interrupts,每个处理器都有一个专属的Timer与WatchDog,支持Level 2 AMBA(AXI high-speed Advanced Microprocessor Bus Architecture)接口,每个处理器都有一个Integral EmbeddedICE-RT Logic用以提供JTAG除错接口,与各自的Pipeline,Branch Prediction with Return Stack,与CoProcessors 14 and 15,每个处理器都有自己的MMU (Instruction and Data Memory Management Units),主要的差异在于处理器对分页的处理不是直接跟单核心架构一样去操作TLB,而是每个处理器都维护自己的Micro TLB,并透过共享的Main TLB同步,每个处理器都有L1 Instruction/Data Cache,每个处理器都具备对外的32-bit Instruction Interface与64-bit Data Cache,每个处理器都支持硬件的Data Cache Coherence,每个处理器都可提供Vector Floating-Point (VFP) Coprocessor 支持

ARM MPCore架构与周边运作示意图

**[](http://loda.hala01.com/wp-content/uploads/2011/06/image002.png)**

**Interrupt Distributor**

MPCore 的架构下会透过 Interrupt Distributor 统一管理MPCore上所有处理器的中断来源,并且依据中断优先级分派中断给个别的处理器.每个中断来源,都可以设定优先级,以及当该中断发生时,哪些处理器要收到该中断要求.在硬件支持上,会确保一个发送给多处理器的中断,一次只有一个处理器在处理.

以Cortex A9为例,Interrupr Distributor支持224个中断来源,每个中断源都有唯一的ID识别(ID0-ID223),有关中断来源分类如下所示

|  |  |
| --- | --- |
| 来源 | 说明 |
| Software Generated Interrupts (SGI) | 可用于Inter-Processor Interrupts (IPI).每个MPCore中的处理器都会有Private Interrupt 范围从ID0到ID15,并且只能由软件触发中断. 中断的优先级,会由每个接收中断的处理器自行设定决定,发出中断的处理器无法决定接收端的优先级. |
| Global timer (PPI(0)) | 透过Interrupt Distributor 使用中断ID27 |
| A legacy nFIQ pin (PPI(1)) | 如果选择Legacy FIQ mode,就会跳过Interrupt Distributor直接把中断发给每个MPCore处理器.  反之,就会藉由Distributed Interrupt Controller把FIQ以中断ID28发给MPCore的处理器. |
| Private timer, PPI(2) | 每个MPCore中的处理器会以中断 ID29作为Private Timer中断源. |
| Watchdog timers, PPI(3) | 每个MPCore中的处理器会以中断 ID30作为Watchdog Timer中断源. |
| A legacy nIRQ pin, PPI(4) | 如果选择Legacy IRQ mode,就会跳过Interrupt Distributor直接把中断发给每个MPCore处理器.  反之,就会藉由Distributed Interrupt Controller把IRQ以中断ID31发给MPCore的处理器. |
| Shared Peripheral Interrupts (SPI) | 用以衔接周边装置中断之用,可设定为Edge Sensitive (posedge) 或 Level Sensitive(high level),并从中断编号ID32开始.(Interrupt Distributor 支援最多224个中断源 ) |

Interrupt Distributor 的Prioritization and Selection功能,会去找出目前最高优先级的Pending中断源(其中: 0x00为最高优先级,0x0f为最低优先级),并将该中断透过CPU Interface进行触发.

Interrupt Distributor 会帮每个处理器维护一个尚未处理的中断列表,并且选择最高优先级的中断发给对应的处理器,若中断优先级相同,则选择最低的中断源编号 (ID0-ID223) 进行触发. 中断列表中会包括:优先级,中断触发的目标处理器.

Interrupt Distributor 支持1-N与N-N两种中断模式,说明如下所示

|  |  |
| --- | --- |
| 中断模式 | 说明 |
| 1-N | 所触发的中断可以被任一的处理器清除,并且其他尚未处理该中断的处理器对该中断的状态也会被清除. |
| N-N | 每个处理器对该中断的处理行为各自独立. 个别处理器对该中断的清除,并不影响到其他尚未处理到该中断处理器的中断状态. |

当收到来自处理器发出的 ‘End of Interrupt Information (EOI)’ ,确认对应中断在处理器已被处理完毕(Active to Inactive transition),或是通知正在进行处理(Pending to Active transition), Interrupt Distributor就会改变所维护的中断清单状态. MPCore处理器的中断可以处于以下三种状态,

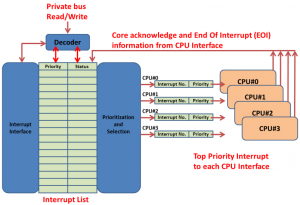
Inactive: 该中断可能尚未被触发,或是已经触发,并且在该处理器中被处理完毕.同时,该中断源也可能在其他处理器中还处于Pending或是Active的状态,会根据每个处理器处理中断的情况而定.

Pending:该中断已发生,但尚未在对应处理器中触发执行.

Active:该中断已经被执行,但尚未执行结束.

当Interrupt Distributor侦测到中断发生时,就会设定对应目标处理器该中断的状态为Pending.如果该中断为Level-Sensitive,有任一MPCore处理器,对该中断还处于Active的状态时,则该中断就不能设定为Pending.如果是Edge-Sensitive,当前一个中断尚未处理完毕,下一个中断又发生时,在MPCore中,对不同处理器可能同时存在Pending与Active的状态.

Interrupt Distributor运作的概念,如下图所示

**[](http://loda.hala01.com/wp-content/uploads/2011/06/image004.png)**

我们可以透过Software Generated Interrupt Register 或 Interrupt Set-Pending Register 触发软件中断,给特定或是一组处理器,Interrupt Distributor对Hardware与Software中断处理的行为完全一致.只是一个来自硬件,一个是透过软件主动触发的. 软件中断可提供在多核心架构下,跨处理器的中断通知机制,包括可以把一个正在WFI状态的处理器唤醒.

MPCore每个处理器的CPU Interface 可支持中断Priority Masking与Preempted中断(让高优先级的中断可以插断当前的中断),一个Pending中的中断,如果通过Priority Mask,并且优先级高于目前处理器正在执行中的Active中断,就会被该MPCore处理器插入执行.当处理器透过Interrupt Acknowledge Register读取目前要处理的中断编号时,CPU Interface就会记录该中断的优先级,并通知Interrupt Distributor将该中断标示为Active. 如果在处理器读取Interrupt Acknowledge Register前,该中断因为Priority Mask更改或是透过Interrupt Pending Clear Register被取消了,则会从Interrupt Acknowledge Register读取到1023,表示没有需要被处理的中断.当中断处理完毕,就会需要处理器设定End of Interrupt Register,用以透过CPU Interface通知Interrupt Distributor将该中断标示为Inactive.

**Cache coherence机制**

在ARM多核心的架构下,每个处理器都会有自己的L1 Cache,并共享同一块L2 Cache,也因为如此,当两个处理器的Cache有暂存到同一个地址的内存数据时,如果没有一个协同确认机制的就有机会导致某个处理器对同一个地址的数据做了修改,但是另一个处理器上读出的却是尚未修改过的内容,如此就会导致系统潜藏的错误问题.

ARM多核心架构下采用的是Tightly Coupled Memory,所有的处理器会共享同一块外部内存,可以更自进行工作安排,由于各自的处理器有自己的L1 Cache与共享的L2 Cache,也因此,外部内存,L2与各处理器中的L1 Cache Coherency就会变得重要.(Loosely-coupled 各处理器就不会共享同一块外部内存,各处理单元可透过Message-Passing机制进行沟通)

Cache Coherence 主要会同步每个处理器L1与L2的Cache,如果有处理器更新到另一个处理器Cache中也有同地址的数据内容时,就会透过这机制把有同地址数据的处理器Cache进行更新,参考有关Cache Coherence的信息例如[**http://en.wikipedia.org/wiki/Cache\_coherency**](http://en.wikipedia.org/wiki/Cache_coherency) .

ARM所采用的Snooping Control Unit主要行为为会监控个别处理器Cache 存取数据的地址,如果有一个写入的动作发生在其他Cache也有复制到的地址的数据,Cache Controller就会把该监控的内存位置设定为失效. Snooping Protocol优点为,速度快,由与所有的Request/Response Transaction都会Broadcast到系统中所有单元,被所有的处理器监控到,缺点就是Snooping Protocol不适合更大型的多核新处理器架构,要不就是必须提高Bus的Bandwidth.

其它的作法还包括Directory-Based Coherence,这是一个目录式的架构,属于每个Cache所共同使用到的数据会被放置在一个共通的目录下,这个目录工作行为就像是一个过滤器,处理器必须要透过该机制才可以把数据从外部内存加载到自己的Cache中. 如果该共享的数据被改变了,目录内容就会同步更新,或是会把相关Cache中的数据设定为失效.缺点就是,数据的取得会有比较长的延迟 (必须有 3 hop流程Request/Respond/Forward),好处是Transaction过程,只需要跟Directory-based Controller同步就好,不需要对系统中所有的单元Broadcast. 一般来说,超过64个处理器单元的架构,就会采用Directory-based.

**Snoop Control Unit**

SCU 主要用以连结1-4个MPCore处理器,透过AXI Bus去存取Memory system,主要功能包括

1,同步每个MPCore处理器的Data Cache内容 **(不包括Instruction Cache的同步)**

2,初始化L2 Cache与AXI Memory Access的行为

3,仲裁每个MPCore处理器对L2 Cache的存取行为

4,管理ACP(Accelerator Coherency Port)接口的存取

ACP(Accelerator Coherency Port)主要用于连接原本不被处理器Cache管理的AXI Master外围,例如:DMA Engine. 过往的设计中,如果有一块内存是会被硬件DMA直接更动内容,则该内存我们就会设定为non-Cached,以避免因为处理器的Cache把该内存内容暂存,但该内存内容在外部内存中却已经被硬件修改了,导致两者不一致的执行正确性问题. 为了规避这问题,选择把对应内存的Cache关闭,带来的缺点就是处理器必须去等待外部相对Cache而言较慢速的内存进行数据的存取,导致效能上的减损. 也因此,透过SCU上的ACP,就可以让硬件DMA更动数据内容时与处理器内部的Cache保持一致性,就算硬件DMA要更动对应内存地址的内容,基于ACP的同步机制,就可以在开启处理器Cache的状态下,确保Cache内容被更新到,让整体运作效能维持在高档.

ACP主要目的为让其它装置也可以共享并存取L1/L2 Cache中的资料内容,以期可以在不增加系统功耗的情况下(减少到外部内存存取的次数)增加系统效能. ACP装置Read/Write的行为如下所示

|  |  |
| --- | --- |
| READ | ACP上的装置,要进行Read动作时,会先确认数据是否有在L1 Cahce,反之,则确认是否有在L2 Cache中,最后才是从外部内存中取得,也就是说如果数据有在L1或是L2 Cache中,ACP上的装置会直接从Cache中抓取数据,加速运作的效率 |
| WRITE | ACP上的装置,要进行Write动作时,会确认L1 Cache中是否有暂存同一地址的数据,若有,会Invalidate L1上的数据,并把该笔更新的数据内容配置到L2 Cache中. |

**WatchDog Reset**

在系统设计时,为了避免软件遇到无法Recovery的错误,WatchDog会是系统最后一到防线,而在MPCore架构下,每个处理器都有自己的WatchDog Counter,一旦该Counter太久没有被踢到,导致Counter倒数为0,就会触发Reset的机制.目前MPCore提供的Reset设定包括 (所有的设定都为 Active LOW)

|  |  |
| --- | --- |
| 名称 | 说明 |
| nSCURESET | 用以Reset MPCore Processor Logic,但并不包括个别MPCore CPU Logic |
| nCPURESET[3:0] | 用以Reset目标MPCore CPU Logic (但不包括CP14 Debug Logic) |
| nWDRESET[3:0] | 用以Reset WatchDog Reset Status Flag,但如果这次系统Reset,是由特定处理器的WatchDog Reset所触发,则对应的Status Flag就不会被重置. |
| nNEONRESET[3:0] |  |
| DBGnTRST | 用以进行DBGTAP reset |
| nDBGRESET[3:0]  or  nPORESET[3:0] | 用以在PowerOn Reset初始化CP14 Debug Logic |
| RESETREQ[3:0] | 可针对目标处理器或是所有的处理器,触发Reset流程. 被用在WatchDog倒数为0时触发. |

基于上述的设定选项,参考Cortex A9 MPCore文件,Reset总共可以包括以下的组合,其中包括整个MPCore的Reset,或是针对个别CPU Logic,以及针对Debug与WatchDog Status Flag的动作.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | nSCURESET and nPERIPHRESET | nCPURESET[3:0] | nNEONRESET[3:0] | nDBGRESET[3:0] | nWDRESET[3:0] |
| Cortex-A9 MPCore  Power on reset | 0 | All 0 | All 0 | All 0 | All 0 |
| Cortex-A9 MPCore  Software reset  (CP14 Debug Logic不做Reset) | 0 | All 0 | All 0 | All 1 | All 0 |
| Per processor  Power on reset | 1 | [n]=0 | [n]=0 | [n]=0 | [n]=0 or All 1 |
| Per processor  Software reset | 1 | [n]=0 | [n]=0 | All 1 | [n]=0 or All 1 |
| SIMD MPE  power on | 1 | All 1 | All 1 | All 1 | All 1 |
| Cortex-A9 MPCore  Debug | 1 | All 1 | All 1 | All 0 | All 1 |
| Per processor  Debug | 1 | All 1 | All 1 | [n]=0 | All 1 |
| Per processor  Watchdog flag | 1 | All 1 | All 1 | All 1 | [n]=0 |

当操作系统核心遇到无法修复的错误时,选择 MPCore Power on reset让系统重置,会是比较好的选择.

**Power Management**

消费性电子最重要的感受就是功耗,ARM MPCore处理器同样提供了Run/Standby/Dormany/Shutdown四种电源模式,如下表所示,Wake-Up速度最快的省电模式是Standby Mode,此时处理器的Logic并没有断电,相关缓存器与CoProcessor的状态都维持住,只有处理器进入Clock Gating的状态. 若希望包括处理器Logic也断电,则进一步把相关状态暂存到处理器TCM中,若Leakage Current过高,也可采用Shutdown模式,但把内存与状态回存到Storage上,在下次Wake-Up时,重新回复系统的正常执行.

|  |  |  |  |
| --- | --- | --- | --- |
| Mode | Cortex Processor Logic | OnChip RAM | Wake-Up |
| Run | Power Up 并且所有Logic都有Clock输入 | Power Up |  |
| Standby | Power Up但除了Wake-Up Logic外,是处于Clock Gating的状态 | Power Up | 此时外部内存会进入Low-Power Mode,ARM CPU会处于WFI (Wait For Interrupt)或WFE(Wait For Event)状态,当处理器有收到中断触发,Debug Request….etc时,就会立刻唤醒.  通常要看SoC本身的Leakage Current,会决定在这状态下底电流的消耗情况. |
| Dormant  (最省电的待机状态) | Power Off | Retention state/voltage | 外部内存会进入Low-Power Mode.中断会透过硬件Wake-Up Module让处理器Power On,此时处理器要重新初始化,在进入Dormant Mode前处理器的缓存器与相关状态会储存在OnChip RAM上,通常会透过一个32bits Register记住在OnChip RAM上WakeUp后要执行的程序代码内存地址,以便醒来后,跳过去执行,恢复进入Dormant Mode前的运作状态. |
| Shutdown | Power Off | Power Off | 执行完整重新启动的流程. |

以cortex A9 MPCore四核心的架构来说,总共可以分出14个Power Domain,包括

1,针对四个Cortex A9 Processor的Power Domain (4)

2,针对四个Cortex A9 Processor Data Engine的Power Domain (4)

3,针对四个Cortex A9 Processor Cache与TLB OnChip RAM的Power Domain (4)

4,一个供SCU(Snoop Control Unit) duplicated TAG RAMs的Power Domain (1)

5,一个其它Logc(例如:SCU Logic,Private Peripherals..etc)的Power Domain (1)

上述Power Domain并不包括SoC中其它外围的Power Domain(例如:USB Phy/Logic,Storage..etc),如果要达到最佳省电效益的话,就必须要考虑相关应用的软件行为,搭配对应的PMIC(Power Management IC)的Power Group来做区分,维持系统中只有必须的处理器与外围是Power On与有Clock Input的状态.

若系统的Standby 或Dormant Current够低,在实际的Android手机上,也有快开(Fast-Boot)机制是透过这类机制设计的,让系统不需要真的走到Shutdown模式,而在下次开机时,又可以非常快的回复到正常运作的模式下.

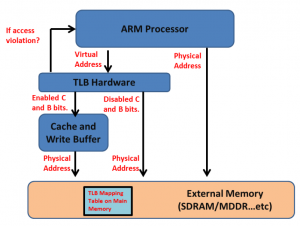
**内存管理系统**

MMU(Memory Management Unit)主要是负责把虚拟内存地址(32-bits 为 0-4GB)对应到物理内存中,而在ARM的架构下负责这层对应机制的就是TLB(Translation Lookaside Buffer). 参考ARM的Technical Reference Manual,在ARM9下,MMU可以支持1MB(Section),64KB(Large Page),4KB(Small Page)与1KB(Tiny Page)四种分页的大小,不过在ARM11与Cortex时,对应的Mapping Size变为4KB, 64KB, 1MB与16MB(Super Section),其中1KB的分页已经不存在了(应该也是因应目前Application Processor内存的实际配置也较大),对分页与物理内存的对应,举以下例子来说明:用户在虚拟内存地址所使用到的4KB Page,在物理内存中就会是连续的物理内存区块,但如果是两个在虚拟内存中连续的4KB Pages,在物理内存中就有可能是分离的两块物理内存区块,只是因为透过硬件MMU TLB的配置,在虚拟内存空间中,应用程序的执行与使用会当成是两个连续的内存区块.

MMU对多任务环境尤其重要,例如在Linux环境中,每个Task都会在自己的虚拟内存空间中对应到一组共享的共享函式库,这些共享函式库就可以透过MMU把每个Task共享的共享函式内容,用同一块物理内存对应到每个Task各自的虚拟内存空间中,如此虽然系统中有多个Task同时运作,但每个Task所共享的内存内容部分就可以透过MMU支持对应到同一块物理内存中,减少Run-Time的物理内存需求.另一个例子就是,Android Dalvik的Java应用,每个MMI的应用程序都会基于一个Dalvik VM运作,同时每个VM也都会加载相关动态函式库.so/.jar,基于MMU的机制,虽然是每个MMI应用都基于一个Dalvik VM,但实际上共享的部份就会透过虚拟内存分页配置对应到同一块物理内存中,降低实际的内存需求.

MMU也可以把内存范围依据User Mode与Kernel Mode设定不同的权限,像是Non-Access,Read Only,Read/Write,如果有发生违反权限的存取行为时,就会导致ARM触发Abort例外.

在ARM平台上,负责查表由虚拟内存对应到物理内存的动作是由硬件TLB机制来实现的,在单一ARM处理器的架构下,通常只有一个TLB分页配置地址,也就是说目前Task Context-Switch到Task #A就会把该Task的TLB 分页配置地址加载,如果是又切换到Task #B就会加载该Task的TLB 分页配置地址.单核心的架构下,同一时间处理器只会处理一个Task,因此TLB的配置只需要符合这样的行为即可. 如下图所示

**[](http://loda.hala01.com/wp-content/uploads/2011/06/image006.png)**

然而,在多核心的架构下,同一时间会有多个Task被执行,因此会在每个核心都提供Micro TLB,用以加载目前该核心正在执行中的Task TLB内存分页,并且整个系统会有一组Main TLB,用以同步每个处理器Micro TLB共享的部份.

以ARMv7架构为例,要控制TLB Translation Table,可以透过CP15的缓存器c2,其中c2主要提供以下Translation table base registers

|  |  |
| --- | --- |
| 名称 | 说明 |
| Translation Table Base Register 0 (TTBR0) | 用来记录User-Mode应用Task的内存分页架构所在的Base Address,通常大小为128bytes到16kbytes (也就是说每个Task的 1st Level Table可以有32到4k笔Items(也就是 1st Level Table Index的最大长度),可透过TTBCR.N值决定),当操作系统进行Context Switch时,会把这个缓存器的值,指到新的Task的内存分页架构的Base Address,并更新TTBCR与CONTEXTIDR缓存器.如果TTBCR设定为0,则以ARMv6以前的架构来操作 TTBR0.(也就是说只有一个 TTBR,User-Mode与Kernel-Mode的内存分页都透过它描述,相对的当Context-Switch发生时,就缺少分出 TTBR0与 TTBR1的弹性). |
| Translation Table Base Register 1 (TTBR1) | 用来记录操作系统特权等级与I/O空间的内存分页架构所在的Base Address,属于这类的内存规划,并不会随着应用Task Context-Switch而改变.通常这Table大小都为16Kbytes. (就是说 1st Level Table可以有最多 4k笔Items(= 1st Level Table Index的最大长度)). |
| Translation Table Base Control Register (TTBCR) | 在没有 Trust Zone Security Extensions支持的环境下,对应的字段如下所示   |  |  | | --- | --- | | 31 – 3 | 2 – 0 | | UNK/SBZP | N |   其中N[2:0] 用以表示TTBR0的宽度,也就是说TTBR0的Base Address Bits数为[31:14-N],如果N=0,表示TTBR0对应的Table大小为14bits=16kbytes,如果N=b111=7,表示TTBT0对应的Table大小为7bits=128bytes. |

可透过如下指令存取

MRC p15,0,<Rt>,c2,c0,0 ; Read CP15 Translation Table Base Register 0

MCR p15,0,<Rt>,c2,c0,0 ; Write CP15 Translation Table Base Register 0

MRC p15,0,<Rt>,c2,c0,1 ; Read CP15 Translation Table Base Register 1

MCR p15,0,<Rt>,c2,c0,1 ; Write CP15 Translation Table Base Register 1

MRC p15,0,<Rt>,c2,c0,2 ; Read CP15 Translation Table Base Control Register

MCR p15,0,<Rt>,c2,c0,2 ; Write CP15 Translation Table Base Control Register

有关Domain控制参数,,可以透过CP15的缓存器c3 并取得 Domain Access Control Register 32-bits的值.DACR缓存器只能在特权等级下被存取,该缓存器如下图所示会以各2 bits被区分为16个字段.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **31** | **30** | **29** | **28** | **27** | **26** | **25** | **24** | **23** | **22** | **21** | **20** | **19** | **18** | **17** | **16** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
| D15 | | D14 | | D13 | | D12 | | D11 | | D10 | | D9 | | D8 | | D7 | | D6 | | D5 | | D4 | | D3 | | D2 | | D1 | | D0 | |

每个字段各2bits的属性意义如下所示  
(参考这个连结 [**http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0434b/CIHBCBFE.html**](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0434b/CIHBCBFE.html))

|  |  |
| --- | --- |
| 域值 | 说明 |
| b00 | No access. Any access generates a domain fault. |
| b01 | Client Mode. 会对每个存取对应TLB内存分页的动作,执行 Access Permission 的检查.若有违反权限的存取行为,就会触发 Permission Fault. |
| b10 | Reserved. Any access generates a domain fault. |
| b11 | Manager Mode. 将部会对存取内存分页的动作进行Access Permission的检查,就算该内存设定为Read-Only或是eXecute Never(XN),去写入或是执行,都不会触发 Permission Fault. |

可透过如下指令存取

MRC p15,0,<Rt>,c3,c0,0 ; Read CP15 Domain Access Control Register

MCR p15,0,<Rt>,c3,c0,0 ; Write CP15 Domain Access Control Register

有关处理器切换的Task ID设置,,可以透过CP15的缓存器c13 取得 Context ID Register (CONTEXTIDR) 32-bits的值. CONTEXTIDR对应字段意义如下所示

|  |  |
| --- | --- |
| 31 — 8 | 7 — 0 |
| PROCID | ASID |

字段说明如下所示

|  |  |
| --- | --- |
| 域名 | 说明 |
| PROCID | 用以储存每个处理器正在执行的Task唯一的标识符. |
| ASIC | Application Space IDentifier  Address Space Identifier 用以记录目前使用的内存空间标识符. |

可透过如下指令存取

MRC p15,0,<Rt>,c13,c0,1 ; Read CP15 Context ID Register

MCR p15,0,<Rt>,c13,c0,1 ; Write CP15 Context ID Register

除了Task ID本身,CP15的c13还提供TPIDRURW (User Read/Write Thread ID Register),

TPIDRURO(User Read-only Thread ID Register),TPIDRPRW(Privileged Only Thread ID Register)分别提供在User-Mode/Kernel Mode Read/Write,User-Mode Read-Only Kernel Mode Read/Write或只有Kernel Mode可以 Read/Write用以设定Thread Id的对应缓存器值.

可透过如下指令存取

MRC p15, 0, <Rt>, c13, c0, 2 ; Read CP15 User Read/Write Thread ID Register

MCR p15, 0, <Rt>, c13, c0, 2 ; Write CP15 User Read/Write Thread ID Register

MRC p15, 0, <Rt>, c13, c0, 3 ; Read CP15 User Read-only Thread ID Register

MCR p15, 0, <Rt>, c13, c0, 3 ; Write CP15 User Read-only Thread ID Register

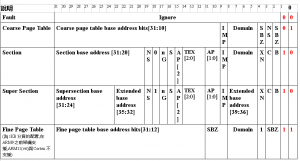
MRC p15, 0, <Rt>, c13, c0, 4 ; Read CP15 Privileged Only Thread ID Register

MCR p15, 0, <Rt>, c13, c0, 4 ; Write CP15 Privileged Only Thread ID Register

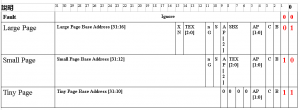
接下来,说明一个32bits虚拟内存是如何透过TLB对应到物理内存的地址,如下表所示,可以区分为三个部分,分别为12bits的第一级Table Index,10bits的第二级Table Index与最后10bits的Page Index,以此细分最多可以把每个Page的大小细分到 1Kbytes,并对应到完整的4GB虚拟内存空间. (2^12 \* 2^10 \* 1kbytes = 4GB.)

|  |  |  |
| --- | --- | --- |
| 31 — 20 (12-bits) | 19 – 10 (10 bits) | 9 – 0 (10bits) |
| 1st Table Index | 2nd Table Index | Page Index |

第一级Table的描述内容如下所示

**[](http://loda.hala01.com/wp-content/uploads/2011/06/Table1.png)**

第二级Table的描述内容如下所示

**[](http://loda.hala01.com/wp-content/uploads/2011/06/Table2.png)**

参考 [**http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211h/Babifihd.html**](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211h/Babifihd.html) , 有关Outer与Inner Cache在有L1跟L2 Cache下的定义为 Inner Cache指的就是 L1 Cache,而Outer Cache指的就是L2 Cache,而在内存分页的设定中,就可以根据L1或L2 Cache的相关行为,决定是否 Cacheable ,Write-Through,Write-Back或是Write-Allocate. 一般而言,L1跟L2的Cacheable与Bufferable行为都会设定成一致,但搭配 TEX,C与B字段,也可以根据系统实际的状况,让L1跟L2 Cahce可以依据系统所需的行为,而作差异化的调整.

对应字段的意义说明如下,

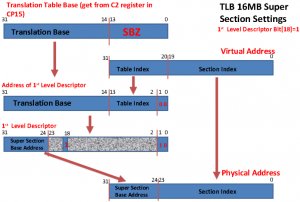
|  |  |
| --- | --- |
| 域名 | 说明 |
| C | Cacheable |
| B | Bufferable |
| TEX[2:0] | TEX  Type Extension (TEX) bit  如下以SCTLR.TRE=0 (透过CP15的c1取得 System Control Register 32bits值的bit 28 来设定),也就是TEX Remap disabled模式,来说明内存区段属性的配置与意义.   |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | | TEX[2] | TEX[1] | TEX[0] | C | B | S | 说明 | Memory Type | Page Shareable | | 0 | 0 | 0 | 0 | 0 | X | Strongly-ordered | Strongly-ordered | Shareable | | 0 | 0 | 0 | 0 | 1 | X | Shareable Device | Device | Shareable | | 0 | 0 | 0 | 1 | 0 | O | Outer and Inner Write-Through, no  Write-Allocate | Normal | Depond on S bit | | 0 | 0 | 0 | 1 | 1 | O | Outer and Inner Write-Back, no Write-Allocate | Normal | Depond on S bit | | 0 | 0 | 1 | 0 | 0 | O | Outer and Inner Non-cacheable | Normal | Depond on S bit | | 0 | 0 | 1 | 0 | 1 | X | Reserved |  |  | | 0 | 0 | 1 | 1 | 0 | X | IMPLEMENTATION DEFINED | IMPLEMENTATION DEFINED | IMPLEMENTATION DEFINED | | 0 | 0 | 1 | 1 | 1 | O | Outer and Inner Write-Back, Write-Allocate | Normal | Depond on S bit | | 0 | 1 | 0 | 0 | 0 |  | Non-shareable Device | Device | Non-shareable | | 0 | 1 | 0 | 0 | 1 |  | Reserved |  |  | | 0 | 1 | 0 | 1 | X |  | Reserved |  |  | | 0 | 1 | 1 | X | X |  | Reserved |  |  | | 1 | **B** | **B** | **A** | **A** | O | AA 用来设定Inner Cache(L1 Cache的参数)  BB 用来设定Outer Cachel(L2 Cache的参数)   |  |  | | --- | --- | | AA或BB的编码值 | 说明 | | 0 0 | Non-cacheable | | 0 1 | Write-Back, Write-Allocate | | 1 0 | Write-Through, no Write-Allocate | | 1 1 | Write-Back, no Write-Allocate | | Normal | Depond on S bit | |
| XN | 为**Execute Never 的属性,**若该内存分页 XN Bit设定为1,表示该分页不会被处理器Fetch指令进来执行,在Client Domain (也就是会稽核 Access Permission状态)下,内存分页必须要 XN Bit为0,且内存属性是设定为可读取,同时没有其他Prefech Abort发生的状态下,才可以被执行. 如果该内存分页是属于Manager Domain,XN Bit就不会被当做稽核的条件. (所以就可以尝试去执行,而不会导致例外发生). |
| NS | 这个属性在支持 Trust Zone Security Extensions环境下,才会有作用. |
| Domain | 参考ARMv7的文件, VMSA() 会以4-bits表示Domain的Index,也就是说最大可以定义到16个Domain,每个Domain Index会依序对应到Domain Access Control Register 32-bits值中以各2-bits依序产生的16个字段. |
| **AP[2], AP[1:0]** | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | AP[2] | AP[1] | AP[0] | **Privileged**  **permissions** | **User**  **permissions** | 说明 | | 0 | 0 | 0 | No Access | No Access | All accesses generate Permission faults | | 0 | 0 | 1 | Read/Write | No Access | Privileged access only | | 0 | 1 | 0 | Read/Write | Read Only | Writes in User mode generate Permission faults | | 0 | 1 | 1 | Read/Write | Read/Write | Full access | | 1 | 0 | 0 |  |  | Reserved | | 1 | 0 | 1 | Read Only | No Access | Privileged read-only | | 1 | 1 | 0 | Read Only | Read Only | Privileged and User read-only, deprecated in VMSAv7 | | 1 | 1 | 1 | Read Only | Read Only | Privileged and User read-only | |
| S | 用以定义该内存分页是否为Shareable,S为 0 表示该内存分页为Non-shareable,S 为 1 表示该内存分页为 Shareable. |
| nG | 这属性为N**on-Global,用来定义该内存分页是否为Global,如果nG为0,表示该内存分页为Global,如果为1,表示该内存分页属于目前正在运作的ASID(Address Space Identifier),该值会对应到正在运作的Task (请参考CONTEXTIDR)** |
| Bit [18] | when bits [1:0] == 0b10  0 Descriptor is for a Section  1 Descriptor is for a Supersection. |
|  |  |

有关3bits AP (Access Permission)与4bits Domain对应的行为,说明如下所示

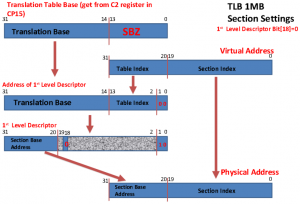
|  |  |
| --- | --- |
| Manager Domain | 若 Domain=1,且参考Domain Access Control Register对应到Domain Field 1的值为0b11,也就是Manage Domain,此时如果,AP[2:0]值为0b000,也就是Privileged/User Mode都是 No Access,由于此时为Manager Domain,因此对任何内存范围的存取都不会进行权限的检查动作,因此不管此时是处于Privileged/User Mode,都可以对该内存内容进行存取的动作. |
| Client Domain | 若 Domain=2,且参考Domain Access Control Register对应到Domain Field 2的值为0b01,也就是Client Domain,此时如果,AP[2:0]值为0b010,也就是Privileged Mode为 Read/Write,User Mode为Read Only.  如果此时CPU处于Privileged Mode,可以对该内存进行读写  若此时CPU处于User Mode,则只允许对该内存进行读取,写入的动作会导致Permission faults |
| Client Domain | 若 Domain=3,且参考Domain Access Control Register对应到Domain Field 3的值为0b01,也就是Client Domain,此时如果,AP[2:0]值为0b001,也就是Privileged Mode为 Read/Write,User Mode为No Access.  如果此时CPU处于Privileged Mode,可以对该内存进行读写  若此时CPU处于User Mode,对该内存的存取动作,都会导致Permission faults |

根据TLB的设定参数组合,接下来分别以基于16MB(SuperSection),1MB(Section),64KB(Large Page),4KB(Small Page)与1KB(Tiny Page)不同分页的组合,来说明TLB 1级与2级 Table的运作概念,

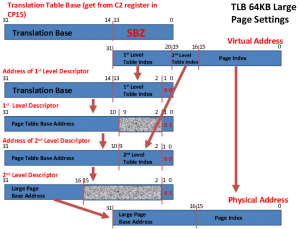
如下所示为16MB(SuperSection) 配置下,TLB分页运作的概念

**[](http://loda.hala01.com/wp-content/uploads/2011/06/image008.png)**

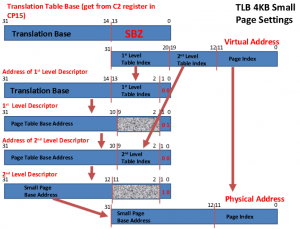
如下所示为1MB(Section) 配置下,TLB分页运作的概念

**[](http://loda.hala01.com/wp-content/uploads/2011/06/image010.png)**

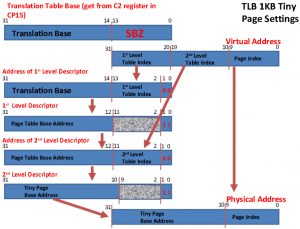
如下所示为64KB(Large Page) 配置下,TLB分页运作的概念

**[](http://loda.hala01.com/wp-content/uploads/2011/06/image012.png)**

如下所示为4KB(Small Page) 配置下,TLB分页运作的概念

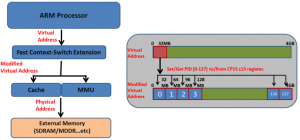
**[](http://loda.hala01.com/wp-content/uploads/2011/06/image014.png)**

如下所示为1KB(Tiny Page) 配置下,TLB分页运作的概念

**[](http://loda.hala01.com/wp-content/uploads/2011/06/image016.png)**

**Fast Context-Switch Extension (FCSE)**

快速行程切换主要支持以32MB内存范围为单位,把Virtual Address转为Modified Virtual Address对应的空间,如下图所示,虚拟内存地址会透过FCSE逻辑,转为修正后的虚拟内存地址,再透过MMU对应到物理内存空间(Physical Memory Address),也就是说,基于FCSE整个系统可以共享同一份虚拟内存空间配置表,不需要在每个Task切换时,更新整个虚拟内存空间配置表,只需要把各自对应的32MB起点位置透过FCSE对应到目前所在虚拟内存空间的对应位置(Modified Virtual Memory Address)即可.

**[](http://loda.hala01.com/wp-content/uploads/2011/06/image018.png)**

FCSE最多只能切割128个32MB内存空间,并对应到4GB的虚拟内存配置,同时,每个Task各自拥有的虚拟内存空间必须是0x00000000起点的内存位置,并依据7bits的PID来对应到各自的32MB内存区块,运作行为可用以下的简式来说明

if(VA[31:25] ==0) then

MVA = VA | (PID<<25)

else

MVA = VA

参考ARM1176JZ-S Technical Reference Manual,ARM为了让早期Windows CE 每个行程各自32MB虚拟内存空间的设计,可以有效的运作,因此提供了快速行程切换(FCSE)机制.当发生Task切换时,可以透过设定CP15 c13 缓存器中FCSE PID的值,决定FCSE所要转换的内存空间,对应带来的好处就是,由于整个系统在Task切换后,还是基于同一份TLB的虚拟内存配置表,同时配置表中的内容仍旧是有效的,只是透过FCSE把运作的Task虚拟32MB内存空间转换到对应的位置,因此,可以减少TLB Flush的成本,减少Task Context-Switch切换的成本.

可透过如下程序代码读/写FCSE PID

MRC p15, 0, <Rd>, c13, c0, 0; Read FCSE PID Register

MCR p15, 0, <Rd>, c13, c0, 0; Write FCSE PID Register

缓存器格式如下所示

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | **0** |
| FCSE PID | | | | | | | SBZ | | | | | | | | | | | | | | | | | | | | | | | | |

参考网页[**http://en.wikipedia.org/wiki/Windows\_Embedded\_CE\_6.0**](http://en.wikipedia.org/wiki/Windows_Embedded_CE_6.0) 的说明 ,在Windows CE 6以前,每个应用程序只能拥有属于各自32MB的虚拟内存空间,并且最多只能有32个Tasks被加载到内存中, 到了Windows CE 6之后的核心,选择支持完整的内存分页机制,提供每个Task各自2GB的虚拟内存空间,并支持最多32768 (2^15)个Tasks被加载到内存中. 对应到Windows Mobile的版本,则是直到Windows Phone 7 才采用Windows CE 6.0 R3的核心,解除了每个应用程序32MB虚拟内存的限制.

ARM在ARM11(ARMv6)并不建议采用FCSE,而在Cortex (ARMv7)方案中,FCSE为一个Optional选项.

**Micro TLB与Main TLB(Translation Lookaside Buffer)架构**

参考相关Two-Level TLB (Micro/Main)架构的文件,为了要减少TLB查询对外部内存存取的成本,因此在On-Chip上增加SRAM,透过高速的内存,Micro TLB可以在1个CPU Clock就完成查询的结果,若Miss,才是到Main TLB查询,若还是Miss,才是到外部内存上的TLB查询. 另外这篇论文 “An Advanced Filtering TLB for Low Power Consumption”(参考网址:[**http://supercom.yonsei.ac.kr/paper/An%20advanced%20filtering%20TLB%20for%20low%20power%20consumption.pdf**](http://supercom.yonsei.ac.kr/paper/An%20advanced%20filtering%20TLB%20for%20low%20power%20consumption.pdf) ),以Strong ARM为例(记忆中频率约为200MHz),在功耗的消耗上,个内存区块大约各占Instruction Cache:27%,TLB(Translation Lookaside Buffer ):17%与Data Cache:16%,由此可知Instruction Cache与TLB在处理器运作时,是相当频繁被存取使用的.同时,当Micro/Main TLB支援笔数增加,虚拟内存查询Miss Rate就会降低,且对外部内存TLB查询的次数减少,也可降低功耗.

TLB本身为一个On-Chip的内存区块,当处理器要去查询外部内存的内存分页表格时,如果要查询的目的内存地址存在于TLB暂存的内存中,就会由TLB直接返回,反之如果该内存位置不在目前TLB暂存的范围中,就会到外部内存分页表格中查询,并且把查询的结果记录到TLB中,以便在下次又查询同样的内存地址时,可以很快速的响应,省去要到外部内存查询的成本.

当外部内存的内存分页表格因为 Task Context-Switch切换而变动时,我们会透过CP15 c2缓存器来修改TLB Base Address,并且Flush TLB快取暂存中的内容(因为已经对应到新的Task内存分页内容),相关Flush的动作可以透过CP15 c8缓存器来执行,如果有些虚拟内存对应的内容并不希望因为Task Context-Switch切换,被TLB Flush所清除,则可以透过CP15 c10缓存器,进行TLB Lock-Down的动作,让该内存对应地址的内容可以保存在TLB暂存内存中,进而加入相关内存转换的速度.

在Cortex A(ARMv7)的Virtual Memory System Architecture 对内存管理部分有如下的演进,

1,内存分页包括4KB,64KB,1MB与16MB.

2,支持内存分页16个Domains

3,支持内存分页Global与ASID(Address space identifiers)机制,避免当Task Context-Switch发生时,TLB被Flush的成本. (属于Global的分页是不属于特定的Task,且ASID 8-bits可用来识别不同Task内存配置的ID)

4,延伸Access Permission的能力,

在Cortex A9中,为了支持多核心的架构,内存管理机制做了以下的演进,

1,支持32笔项目的Instruction Micro TLB

2,支持32笔项目的Data Micro TLB

3,支持一致的Main TLB

4,支持L1 Data Cache的Page Table查询硬件. (可以用来确保每个处理器L1 Data Cache共享的Page一致性.)

在多核心的架构下,ARM为了减少每个处理器出去外部TLB查询Table的次数,因此提供了2 Level的TLB架构(Micro与Main TLB),

|  |  |
| --- | --- |
| Micro TLB | 每个处理器都会有自己的Micro TLB,提供Instruction与Data各32笔对应的项目(ARM11 MPCore为各八笔),可以在1个CPU Cycle把Virtual Address转为Physical Address,包括该内存分页的保护属性,或是否要触发Prefech/Data Abort.如果该Virtual Address不在这Instruction或Data TLB 的纪录中,就会往Main TLB查询.  如果外部的Main TLB有更新,或是发生Task Context-Switch 更新Context ID Register都会导致Micro TLB被Flush. |
| Main TLB | Main TLB可以对应到16MB,1MB,64KB与4KB不同的内存分页,一旦所要查询的Virtual Address在Micro TLB中查询不到,就会到Main TLB中查询. Main TLB为2 ways的配置,可以为2\*32的64笔项目的Main TLB,或2\*64的128笔项目的Main TLB. 每个Main TLB项目都会包括  1,Virtual Address  2,Page Size  3,Physical Address  4,Memory Properties  每一组上述的项目,都会对应到一个特定的Application空间,或是Global让全系统共享,每个ARM核心都会透过设定CONTEXIDR缓存器,记录目前该核心所运作的Application空间,每次TLB项目比对时,只要符合以下条件就算是对比成功  1,虚拟内存跟TLB项目一致 (bits长度可为[31:N],要视内存分页大小而定)  2,Non-secure TLB ID (NSTID)跟目前Secure状态一致. (在有支持Trust Zone Security Extension的环境)  3,跟TLB中的ASID跟目标Task ASID一致或是为Global数性.  其它有关Main TLB lockdown 相关缓存器,如下所示  MCR p15,5,<Rd>,c15,c4,2 #Select Lockdown TLB Entry for Read (Main TLB Index)  MCR p15,5,<Rd>,c15,c4,4 #Select Lockdown TLB Entry for Write (Main TLB Index)  MRC p15,5,<Rd>,c15,c5,2 #Read Lockdown TLB VA Register Data  MCR p15,5,<Rd>,c15,c5,2 #Write Lockdown TLB VA Register Data  MRC p15,5,<Rd>,c15,c6,2 #Read Lockdown TLB PA Register Data  MCR p15,5,<Rd>,c15,c6,2 #Write Lockdown TLB PA Register Data  MRC p15,5,<Rd>,c15,c7,2 #Read Lockdown TLB attributes Register Data  MCR p15,5,<Rd>,c15,c7,2 #Write Lockdown TLB attributes Register Data |

**结语**

随着Smart Phone与平板市场的风行,ARM MPCore架构绝对会是最受瞩目的方案,尤其,包括Android上用NDK开发的应用,或是其它基于ARM平台的方案,都让ARM处理器累积了大量专属的应用与越来越难以取代的角色.

本文主要以笔者所需的信息为主来汇整,对ARM MPCore有更进一步需求的开发者,请自行参阅ARM MPCore技术文件.