第3章 Zynq-7000应用处理单元

Zynq-7000 应用处理单元是该器件最重要的功能单元。只有掌握了应用处理单元的结构原理，才能高效地实现基于 Zynq-7000 的嵌入式系统设计。

3.1 应用处理单元

应用处理单元(Application Processing Unit，APU)存在于 PS 内，包含带有 NEON 协处理器的两个 Cortex-A9 处理器。在多处理器配置中，将两个处理器连接起来共享一个 512KB L2 高速缓存。每个处理器是一个高性能、低功耗的核，各自有两个独立的 32KB L1 数据高速缓存和指令高速缓存。Cortex-A9 处理器实现 ARVV7-A 结构，支持完整的虚拟存储器，能执行 32 位 ARM 指令、16 位及 32 位 Thumb 指令和在 Jazelle 状态下的一个 8 位 Java 字节码。NEON 协处理器内的媒体和信号处理结构增加了用于音频、视频、图像和语音处理和 3D 图像的指令。这些高级的单指令多数据流(Single Instruction Multiple Data，SIMD)指令可用于 ARM 和 Thumb 状态。

3.1.1 基本功能

图 3.1 给出了 APU 的块图结构。多核配置的两个 Cortex-A9 处理器带有一个侦测控制单元 SCU，用于保证两个处理器之间，以及与来自 PL 的 ACP 接口一致性。为了提高性能，Cortex-A9 多核提供了一个用于指令和数据的，共享统一的 512KB L2 高速缓存。与 L2 高速缓存并列，提供了一个 256KB 的片上存储器 OCM，用于提供一个低延迟的存储器。

加速器一致性端口 ACP，用于方便 PL 和 APU 之间的通信。这个 64 位的 AXI 接口允许 PL 作为 AXI 主设备。该设备能访问 L2 和 OCM，同时保证存储器和 CPU L1 缓存的一致性。

统一的 512KB L2 高速缓存是一个 8 路组关联结构，允许用户基于缓存行、路或者主设备锁定缓存行的内容。所有通过 L2 缓存控制器的访问能连接到 DDR 控制器，或送到 PL，或 PS 内的其他相关地址的从设备。为了降低到 DDR 存储器的延迟，提供了一个从 L2 控制器

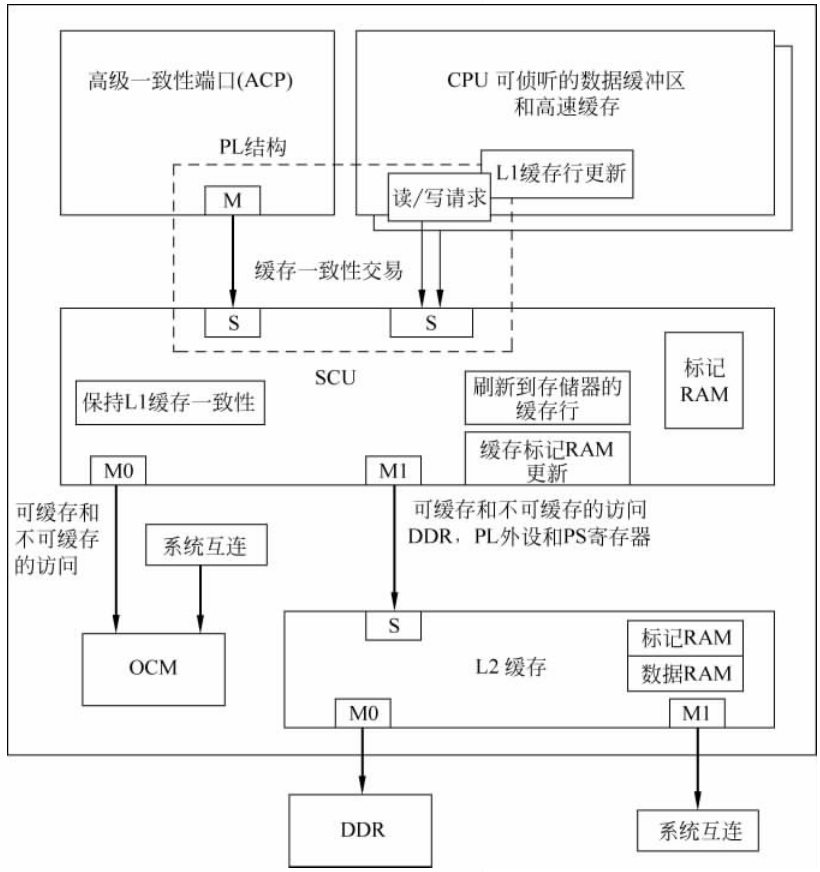

图 3.1　APU 块图

到 DDR 控制器的专用端口。

两个处理器核内建了调试和跟踪能力，并且互联作为 CoreSight 调试和跟踪系统的一部分。用户能通过调试器访问端口(Debug Access Port，DAP)控制和查询所有的处理器和存储器。通过片上嵌入跟踪缓冲区(Embedded Trace Buffer，ETB)或者跟踪端口接口单元(Trace-Port Interface Units，TPIU)，将来自两个处理器的 32 位 AMBA 跟踪总线(AMBA Trace Bus，ATB)主设备和其他 ATB 主设备(如 ITM 和 FTM)汇集在一起，产生统一的 PS 跟踪。

ARM 结构支持多个操作模式，包括超级、系统和用户模式，用于提供不同级别的保护和应用程序级别。这个接口支持 TrustZone 技术，用于帮助创建安全的环境，用于运行应用程序和保护这些应用程序的内容。内建在 ARM CPU 和其他外设的 TrustZone 使能一个安全的系统，用于管理密钥、私有数据和加密信息，不允许将这些秘密泄露给不信任的程序或者用户。

APU 包含一个 32 位的看门狗定时器和一个带有自动递减特性的 64 位全局定时器。它们能用作一个通用的定时器，也可以作为从休眠模式下唤醒处理器的一个机制。

思考题 3-1：请根据前面的介绍，说明 ARM Cortex-A9 双核处理器结构的特点。其高性能主要体现在哪些方面？

3.1.2 系统级视图

APU是系统中最关键的部件，它将PS、PL内所实现的IP、外部存取器和外设这样的板级设备连在一起。图3.2给出了APU的系统结构图。通过L2控制器的两个接口和一个到OCM的接口（OCM和L2高速缓存并列），APU和系统剩余的部分进行通信。

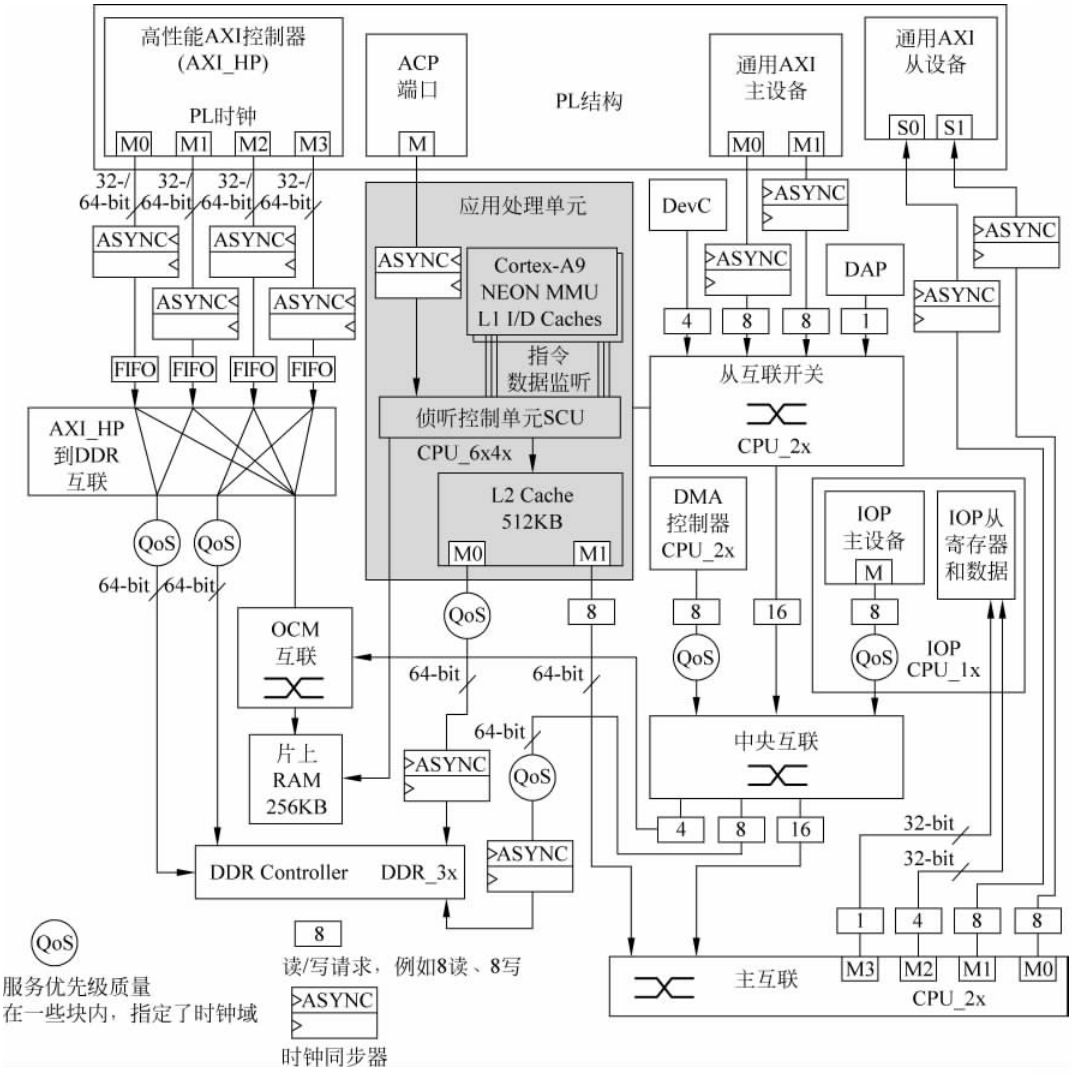

图3.2 APU的系统结构

来自通过SCU的双核Cortex-A9 MP系统的访问，以及来自要求和Cortex-A9 MP系统一致的其他主设备的所有访问，需要使用ACP端口连接到SCU。所有不通过SCU的访问和CPU是非一致性的，软件必须明确地管理同步和一致性。

来自APU的访问，其目标可以是OCM、DDR、PL、IOP从设备或者PS子模块内的寄存器。为了将访问OCM的延迟降低到最小，来自SCU的一个专用主设备端口提供了

处理器和 ACP 到 OCM 地直接访问，其访问延迟甚至小于 L2 高速缓存。

所有到 DDR 的 APU 访问，通过 L2 缓存控制器进行连接。为了改善 DDR 访问延迟，提供了一个来自 L2 缓存控制器到 DDR 存储器控制器的专用主端口，允许所有的 APU-DDR 的交易不经过和其他主设备共享的主互联。来自 APU 的其他访问，这些访问既不绑定 OCM，也不绑定 DDR，通过 L2 控制器。并且，使用第二个端口连接到主互联。通过 L2 缓存控制器的访问是不必缓存的。

如图 3.2 所示，APU 和它的子模块工作在 CPU_6x4x 时钟域。APU 到 OCM 的接口和到主互联的接口都是同步的。主互联能运行在 1/2 或者 1/3 的 CPU 频率。DDR 模块运行在 DDR_3x 时钟域，与 APU 是异步的。到 APU 模块的 ACP 端口包含一个同步器，PL 主设备使用这个端口，使得时钟和 APU 是异步的。

思考题 3-2：互联是 APU 非常重要的结构特点，请说明 APU 内包含哪些互联结构及其各自的特点。

3.2 Cortex-A9 处理器

APU 实现双核 Cortex-A9 MP 配置。每个处理器有自己的 SIMD 媒体处理引擎 NEON、存储器管理单元(Memory Management Unit，MMU)和独立的 32KB L1 指令和数据高速缓存。每个 Cortex-A9 处理器提供了两个 64 位的 AXI 主接口用于到 SCU 的独立指令和数据交易。取决于地址和属性，这些交易连接到 OCM、L2 高速缓存、DDR 存储器或者通过 PS 互联到 PS 内其他的从设备，或者互联到 PL。每个带有 SCU 的处理器接口包含所要求的侦测信号，用于提供处理器内的 L1 数据缓存和用于共享存储器的共享 L2 缓存的一致性。Cortex-A9 和它的子系统也提供了完整的 Trustzone 扩展，用于用户的安全性。

Cortex-A9 处理器实现必要的硬件特性，用于程序调试和跟踪调试生成支持。处理器也提供了硬件计数器，用于处理器和存储器系统操作时搜集统计数据。

Cortex-A9 内的主要子模块包括中央处理单元 CPU、L1 指令高速缓存和数据高速缓存器、存储器管理单元、NEON 协处理器和内核接口。

3.2.1 中央处理器

图 3.3 给出了 Cortex-A9 处理器的内核结构。每个 Cortex-A9 的 CPU 能在一个周期给出两个指令，并且以无序的方式执行。CPU 实现动态地分支预测和可变长度的流水线，性能达到 2.5DMIPs/MHz。Cortex-A9 处理器实现 ARM v7-A 的结构、支持充分的虚拟存储器、能执行 32 位的 ARM 指令、16 位及 32 位的 Thumb 指令和在 Jazelle 状态下的一个 8 位 Java 字节码。

1. 流水线

Cortex-A9 CPU 内所实现的流水线，采用了高级取指和指令预测技术，它将潜在的存储器延迟引起的指令停止和分支解析分开。在 Cortex-A9 CPU 中，预加载最多四个指令缓存行，用于降低存储器延迟对指令吞吐量的影响。CPU 取指单元能在每个周期连续

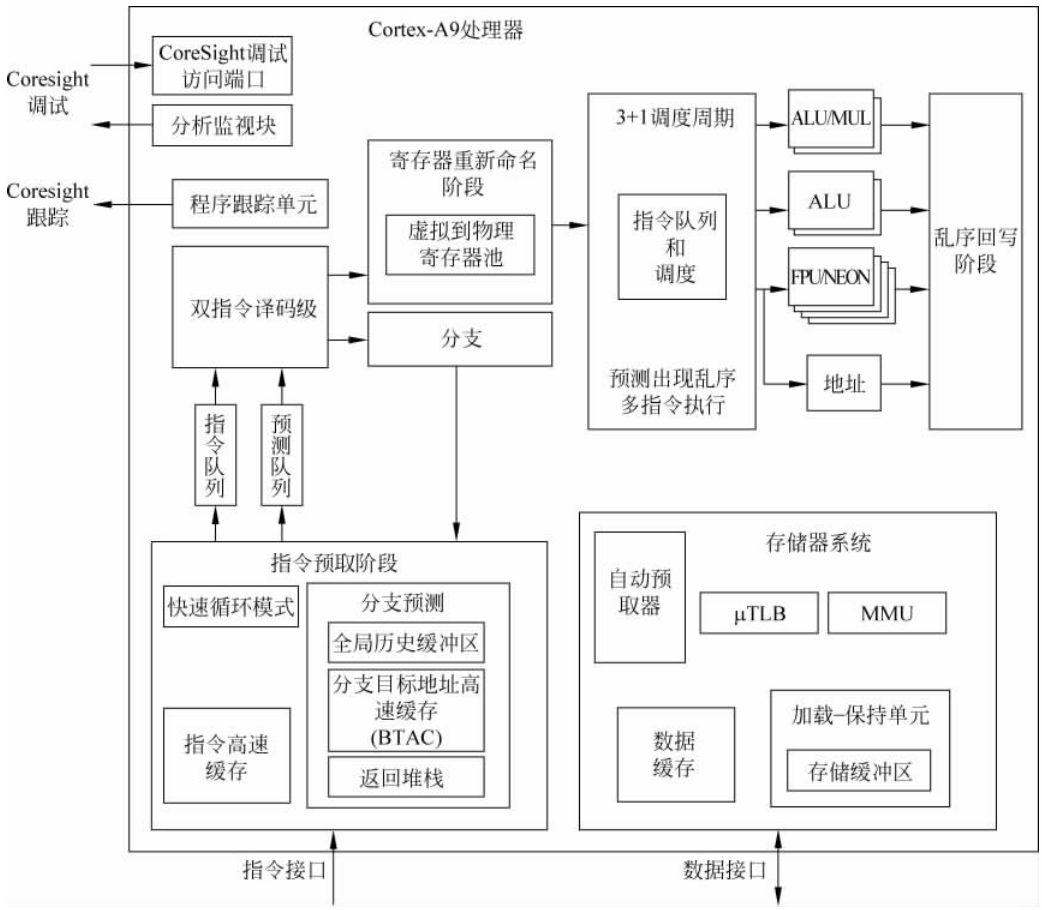

图 3.3 Cortex-A9 处理器的内核结构

发送 2～4 条指令到指令译码缓冲区，以保证高效地使用超标量流水线。CPU 实现一个超标量解码器，能在一个周期内解码两个完整的指令。四个 CPU 流水线中的任何一个流水线都能从发送队列中选择指令。并行流水线支持每个周期贯穿下面单元的并行执行：全部两个算术单元、加载-保存单元和任何分支的解析单元。

Cortex-A9 CPU 采用了预测地执行指令，这样使能动态地重命名物理寄存器到一个虚拟可用的寄存器池中。CPU 使用这个虚拟寄存器重命名来消除寄存器之间的依赖性，但不会影响到程序正确地执行。这个特性通过一个基于循环展开的有效硬件，允许代码的加速。同时，通过在相邻指令间消除数据的依赖性提高流水线的利用率。

Cortex-A9 CPU 中的存储器系统内，提交相互依赖的加载-存储指令用于解析。这样，大大降低了流水线的停止。通过自动或者用户驱动地预取操作，Cortex-A9 CPU 核支持最多四个数据缓存行填充要求。

CPU 的一个关键的特性就是指令的无序写回。这样，就可以释放流水线资源，而不依赖于系统提供的、所要求数据的顺序。

在指令条件或者前面分支解析之前，或者需要写的数据可用之前，能预测地发出加载/保存指令。如果用于执行加载/保存的条件失败，可能产生的任何不利影响，例如刷新修改寄存器的行为。

思考题 3-3：ARM Cortex-A9 双核处理器内采用了流水线结构，其特点主要体现在哪些方面？

2. 分支预测

为了减少在高度流水的 CPU 内分支所造成的不利影响，Cortex-A9 内实现静态和动态地分支预测。由指令提供静态分支预测，在编译时候确定。动态地分支预测使用前面一个指定指令的执行结果，以确定是否采用分支。动态分支预测逻辑使用一个全局分支历史缓冲区(Global Branch History Buffer，GHB)。GHB 是一个 4096 入口的表，包含用于指定分支的 2 位预测信息。当每次执行分支时，更新预测信息。

分支执行和整体的指令吞吐量也得益于分支目标地址缓存(Branch Target Address Cache，BTAC)的实现。BTAC 保存着最近分支的目标地址。这个 512 入口的地址缓存的结构是 2 路×256 入口。基于计算的有效地址和转换的物理地址，在产生真正的目标地址前，用于指定分支的目标地址提供给预加载单元。此外，如果一个指令循环适配四个 BTAC 入口，关闭指令缓存访问，以降低功耗。

Cortex-A9 CPU 能预测条件分支、无条件分支、间接分支、PC 目的数据处理操作和在 ARM 和 Thumb 状态之间切换的分支。然而，不能预测下面的分支指令：

(1) 在状态之间切换的分支(除了 ARM 到 Thumb 转换和 Thumb 到 ARM 转换以外)；

(2) 当它们用于从异常返回时，不能预测带有 S 后缀的指令，因为它们可能改变特权模式和安全性状态，对程序的执行有不利的影响；

(3) 所有用于改变模式的指令。

通过将 CP15 c1 控制寄存器的 Z 比特设置为 1，使能程序流预测。在打开程序流预测前，必须执行一个 BTAC 刷新操作。其额外的效果是将 GHB 设置为一个已知状态。

Cortex-A9 也用一个 8 入口的返回堆栈缓存，保存了 32 位子程序的返回地址。这个特性大大降低了执行子程序调用带来的不利影响，可以寻址最大 8 级深度的嵌套例程。

思考题 3-4：分支预测是指什么？在 Cortex-A9 双核处理器内采用了什么分支预测策略？

3. 指令和数据对齐

ARM 的结构指定了 ARM 指令为 32 位宽度，要求其为字对齐方式。Thumb 指令是 16 位宽度，要求半字对齐。Thumb-2 指令是 16 位或者 32 位宽度，也要求半字对齐。数据访问可以是非对齐的，CPU 内的保存/加载单元将其分解为对齐地访问。当要求的时候，将来自这些访问的数据插入并发送到 CPU 内的寄存器文件中。

注：应用处理单元 APU 和 PS 整体，只支持用于指令和数据的小端结构。

4. 跟踪和调试

Cortex-A9 处理器实现 ARM v7 调试结构。处理器的调试接口由下面构成：

(1) 一个基准 CP14 接口，用于实现 ARM v7 调试结构和 ARM 结构参考手册上所描述的一套调试事件；

(2) 一个扩展的 CP14 接口，实现这个处理器指定的一套调试事件(ARM 结构参考

手册上进行了解释)；

(3) 通过一个调试访问端口 DAP,一个外部的调试接口连接到外部的调试器。

Cortex-A9 包含一个程序跟踪模块,该模块提供了 ARM CoreSight 技术,用于其中一个 Cortex-A9 处理器程序流跟踪能力,并且提供了观察处理器真实指令流的能力。Cortex-A9 程序跟踪宏(Programm Trace Module,PTM)使得对所有代码分支和带有周期计数使能统计分析的程序流变化可见。PTM 模块和 CoreSignt 设计工具使得软件开发者,能够非强制性地跟踪多个处理器的执行历史。并且,通过标准的跟踪接口,将带有校正时间戳的这些历史信息保存到片上缓冲区或者片外存储区。这样,就提高了开发和调试过程的可视性。

Cortex-A9 处理器也实现程序计数器和事件监控器,用于搜集处理器和存储器操作时的统计信息。

3.2.2 L1 高速缓存

两个 Cortex-A9 处理器中的每一个处理器,都有独立的 32KB L1 指令高速缓存和数据高速缓存。L1 缓存的公共特性包括：

(1) 使用系统控制协处理器,能独立地禁止每个缓存；

(2) 所有 L1 缓存的缓存行长度为 32 个字节；

(3) 所有的缓存是 4 路组关联结构；

(4) L1 缓存支持 4KB、64KB、1MB 和 16MB 的虚拟存储器页；

(5) 两个 L1 缓存不支持锁定特性；

(6) L1 缓存有 64 位的接口与整数核和 AXI 主接口相连；

(7) 缓存替换策略为伪轮询或者伪随机,在缺失时,读取淘汰计数器,如果没有分配,则在分配时,递增,在组中,优先使用淘汰计数器替换一个无效的行；

(8) 当缓存缺失时,首先执行关键字填充缓存；

(9) 为了降低功耗,利用许多缓存操作为连续性质的优势,减少读全部缓存的次数,如果一个缓存读和前面的缓存读是连续的,并且是在一个相同的缓存行内读取,则只访问之前所读取的数据 RAM 组；

(10) L1 缓存支持奇偶校验；

(11) 所有存储器的属性输出到外部存储器系统；

(12) 支持 TrustZone 安全性,将安全或者不安全的状态输出到缓存和存储器中；

(13) 在复位时,清除 L1 缓存内容,以遵守安全性要求。

注：*用户在使用指令缓存、数据缓存和 BTAC 前,必须使它们无效。即使为了安全性的原因,也不要求无效主 TLB。这样保证兼容未来处理器的版本。*

L1 指令一侧的缓存负责给 Cortex-A9 处理器提供一个指令流。L1 缓存直接和预取单元接口。预取单元包含一个两级预测机制。L1 指令缓存为虚拟索引和物理标记。

L1 数据一侧的缓存负责保留 Cortex-A9 处理器所使用的数据。L1 数据缓存的关键特性包括：

(1) 数据缓存为物理索引和物理标记；

(2) 数据缓存是非阻塞的,因此加载/保存指令能连续地命中缓存,同时执行由于先

前读/写缺失所产生的来自外部存储器的分配，数据缓存支持 4 个超前地读和 4 个超前地写；

(3) CPU 能支持最多 4 个超前的预加载指令，然而明确的加载/保存指令有较高的优先级；

(4) Cortex-A9 加载/保存单元支持预测的数据预加载，用于监视程序顺序的访问，在请求开始前开始加载下一个期望的行，使用 cp15 辅助控制寄存器(DP 位)，使能这个特性，在分配前可以不使用这个预取行，预加载指令有较高的优先级；

(5) 数据缓存支持两个 32 字节行填充的缓冲区和一个 32 字节的替换(淘汰)缓冲区；

(6) Cortex-A9 CPU 有一个带 64 位槽和数据合并能力的保存缓冲区；

(7) 所有数据读缺失和写缺失是非阻塞的，支持最多 4 个超前数据读缺失和 4 个超前数据写缺失；

(8) APU 数据缓存使用 MESI 算法，完整地侦听一致性控制；

(9) Cortex-A9 内的数据缓存包含本地保存/加载互斥监视程序，用于 LDREX/STREX 同步，这些指令用于实现信号量，互斥监控程序只管理带有 8 个字或者一个缓存行颗粒度的一个地址，因此避免交错的 LDREX/STREX 序列，并且总是执行一个 CLREX 指令，作为任何上下文切换的一部分；

(10) 数据缓存只支持写回/写分配策略，不实现写通过和写回/非写分配策略；

(11) 对于 L2 缓存，L1 数据缓存支持互斥操作，互斥操作是指只有在 L1 或者 L2 内的一个缓存行是有效的，一行填充到 L1，引起该行在 L2 内标记为无效，同时淘汰 L1 中一行，将使得该行分配到 L2 中(即使它不是“脏”的)，来自 L2 脏的一行填充到 L1 时，将淘汰该行到外部存储器中，默认时禁止互斥操作，这样使得增加缓存的利用率和减少功耗。

思考题 3-5：请说明 Cortex-A9 的 L1 结构特点，以及替换策略。

3.2.3 存储器管理单元

ARM 结构中的存储器管理单元(Memory Management Unit，MMU)主要负责存储器保护和地址转换。在虚拟地址到物理地址的转换过程中，MMU 与 L1 和 L2 存储器系统一起密切地工作。它也控制对外部存储器的访问，以及接受来自外部存储器的访问。

MMU 和虚拟存储器系统结构版本 7(VMSAv7)兼容，要求支持 4KB、64KB、1MB 和 16MB 页表入口和 16 个访问域。该单元提供了全局和应用程序指定的标识符，因此免除在上下文切换时需要对 TLB 进行刷新地操作；同时，提供了用于扩展许可检查的能力。

处理器实现 ARMv7-A MMU，提供了扩展的安全性和多处理器扩展。这种扩展提供地址转换和访问许可检查。MMU 控制表搜索硬件，该硬件用于访问主存内的转换表。通过一组虚拟地址到物理地址映射，以及保存在指令和数据转换表 TLB 内的存储器属性，MMU 提供了对存储器系统更好的颗粒度控制。

MMU 负责下面的操作：

(1) 检查虚拟地址和地址空间标识符(Address Space Identifer，ASID)；

(2) 检查区域访问许可；

(3) 检查存储器属性；

(4) 虚拟地址到物理地址转换；

(5) 支持 4 个页面(区域)大小；

(6) 访问缓存或者外部存储器映射；

(7) 可锁定主 TLB 内的 4 个入口。

图 3.4 给出了 MMU 的块图结构。

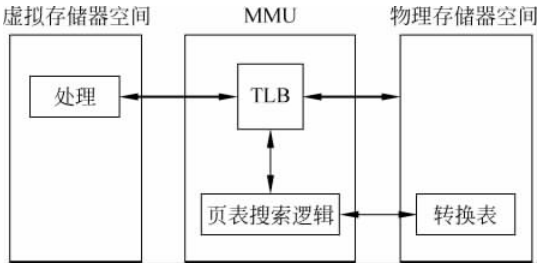

图 3.4 MMU 块图结构

1. 存储器访问顺序

当处理器产生一个存储器访问后，MMU 执行下面的步骤：

(1) 在相关的指令或者数据 microTLB 中，查找于所要求虚拟地址、当前地址空间标识符和安全状态；

(2) 如果在 microTLB 中缺失，则在主 TLB 内，查找所要求的虚拟地址、当前地址空间标识符和安全状态；

(3) 如果在主 TLB 中缺失，则执行一个硬件转换表搜索。

MMU 可能没找到全局映射，或者用于当前所选择地址空间标识符的映射，这个空间标识符带有一个匹配的用于 TLB 内虚拟地址的不安全 TLB ID(NSTID)。在这种情况下，如果使能 TLB 控制寄存器硬件内的 PD0 和 PD1 位，则进行一个转换表的搜索。否则，如果禁止，则处理器返回一个部分转换故障。

如果 MMU 找到一个匹配的 TLB 入口，它使用入口内的如下信息：

(1) 访问许可位和区域决定是否使能访问。如果匹配入口没有通过许可检查，MMU 发出一个存储器退出信号。

(2) 在 TLB 入口和 CP15 c10 重映射寄存器内指定的存储器区域属性，控制缓存和写缓冲区。并且决定访问是：①安全或不安全的；②共享或者非共享；③普通的存储器、设备或者强顺序的。

(3) MMU 将虚拟地址转换为物理地址，用于存储器访问。

如果 MMU 没有找到匹配的入口，则产生硬件表搜索。

2. TLB 的结构和匹配

Cortex-A9 MMU 包含两级 TLB，包含用于指令和数据统一的 TLB 和用于每一个的单独的 micro TLB。第 1 级 TLB 的 micro TLB，每个都有 32 个全关联的入口。如果在一个相应的 micto TLB 内，缺失一个取指或者加载/保存地址，则访问统一的主 TLB。统

一的主 TLB 提供了一个 2 路关联的 2×64 入口的表(128 个入口)。通过使用入口锁定模型,支持 4 个可锁定的入口。TLB 使用一个伪轮询替换决策策略。当出现缺失时,决定应该替换 TLB 中的哪个入口。

不像其他处理器要求软件负责更新驻留在主存内的页转换表 TLB,Cortex-A9 支持硬件页表的搜索,以执行 L1 数据缓存内的查找。这样允许缓存页表。

通过设置转换表基地址寄存器(Translation Table Base Register,TTBR)内 IRGN 位,可以配置 MMU 执行可缓存区域内的硬件转换表搜索。如果对 IRGN 比特编码是写回,则执行一个 L1 数据缓存查找。这样,从数据缓存读取数据。如果对 IRGN 比特编码是写通过或者非缓存,则访问外部存储器。

TLB 入口可以是全局的,或者通过使用与那些进程相关的 ASID,将 TLB 入口分配到一个特殊的进程或者应用程序。ASID 使得在上下文切换时,保持驻留 TLB 入口,以避免要求随后重新加载它们。

注:ARM Linux 内核,不是基于每个 CPU,而是贯穿所有的 CPU 全局来管理 8 位的 TLB ASID 空间。对 ASID 递增,用于每个新的进程。当回卷 ASID(ASID=0)时,TLB 的刷新请求发送给所有的 CPU。然而,只有在一个上下文切换中间的 CPU 立即更新它当前 ASID 上下文,而其他的 CPU 使用它当前回卷前的 ASID 继续运行,直到发生一个调度间隔为止。随后,上下文切换到一个新的进程。

通过集成在核内的专门协处理器-CP15,控制 TLB 的维护和配置。这个协处理器提供一个标准的机制,用于配置 L1 存储器系统。

每个 TLB 入口包含一个虚拟地址、一个页面大小、一个物理地址和一套存储器属性。标记每个 TLB 入口正在关联一个特殊的应用程序空间,或者其作为一个全局用于所有的应用程序空间。如果与被修改的虚拟地址(Modified Virtual Address,MVA)的比特[31:N] 匹配,则与一个 TLB 入口匹配。其中,$N=\log_2$(页面大小)。它或者标记为全局,或者 ASID 匹配当前的 ASID。

当下面条件为真时,匹配一个 TLB 入口:

(1) 它虚拟的地址,匹配所要求的地址;

(2) 它不安全的 TLB ID(NSTID)匹配 MMU 请求的安全或者不安全状态;

(3) 它的 ASID 匹配当前的 ASID,或者它是全局的。

在任意时刻,操作系统必须保证大多数情况下匹配一个 TLB 入口。基于下面的块大小,一个 TLB 可以保存入口:

(1) Supersections:16MB 存储器块。

(2) Sections:1MB 存储器块。

(3) Large pages:64KB 存储器块。

(4) Small pages:4KB 存储器块;

支持 Supersections、Sections 和 Large pages,只使用 TLB 内的一个单个的入口,允许映射大的存储区域。如果在 TLB 内没有找到地址映射,则硬件自动读转换表,将映射放在 TLB 内。

思考题 3-6:请说明 Cortex-A9 双核处理器的存储器管理单元的结构和功能。

思考题 3-7:请说明 Cortex-A9 双核处理器的 TLB 的结构和替换策略。

3.2.4 接口

1. AXI和一致性接口

每个Cortex-A9处理器提供两个64位的伪AXI主接口，用于独立的指令加载和数据交易。这些接口工作时，与处理器核(CPU_6x4x时钟)具有相同的速度。当复制数据穿过一个缓存的存储器区域时，能在每5个处理器时钟周期支持4个双字写操作。指令侧接口是只读接口，没有写通道。这些接口实现一个扩展版本的AXI协议，它提供了到L2缓存的多重优化，其中包括支持L2预取提示和预测的存储器访问。

基于它们的地址，AXI交易通过SCU连接到OCM或者L2缓存控制器。每个Cortex-A9也提供了一个到SCU的高速缓存一致性总线(Cache Coherency Bus，CCB)，用于提供L1和L2缓存间的所要求的一致性管理信息。

2. 调试和跟踪接口

每个Cortex-A9处理器有一个标准的32位APB从接口，该接口工作在CPU_1x时钟频率。通过SOC调试模块内的调试APB总线主设备，访问这个接口。

Cortex-A9处理器也包含一对接口，用于跟踪产生和交替触发控制。来自每个核的跟踪源接口是一个32位CoreSight标准的ATB主接口，其工作在PS互连的速度(CPU_2x Clock)下，该接口连接到SOC调试模块的漏斗形通道上。每个核有一个4位标准的CoreSight交叉触发器接口，其工作在互连时钟频率(CPU_2x Clock)上，该接口连接到SOC调试模块内的交叉触发器开关阵列(Cross Trigger Matrix，CTM)上。

3. 其他接口

每个Cortex-A9处理器有多个控制位，这些位由系统级控制寄存器(System Level Control Register，SLCR)驱动。其中包括：

(1) 一个4位的接口，用于驱动CoreSight标准安全信号；

(2) 用于控制CP15和软件的可编程性的静态配置信号。

3.2.5 NEON

Cortex-A9 NEON MPE扩展了Cortex-A9的功能，提供了对ARM v7高级SIMD和向量浮点(VFPv3)指令集的支持。Cortex-A9 NEON MPE支持所有的寻址模式和数据处理操作。Cortex-A9 NEON MPE的特性主要包括：

(1) SIMD向量和标量单精度浮点计算：①无符号和有符号整数；②单比特系数多项式；③单精度浮点值。

(2) NEON协处理器支持的操作：①加法和减法；②带有可选累加的乘法；③最大或者最小值驱动通道选择操作；④平方根倒数的估计；⑤广泛的数据结构加载指令，包含寄存器组驻留表查找。

(3) 标量双精度浮点计算。

(4) SIMD 和标量半精度浮点转换。

(5) 8 位、16 位、32 位和 64 位有符号和无符号整数 SIMD 计算。

(6) 用于单比特系数的 8 位或 16 位多项式计算。

(7) 结构化数据的加载能力。

(8) Cortex-A9 处理器执行两个 ARM 或者 Thumb 指令。

(9) 独立的流水线用于 VFPv3 和高级 SIMD 指令。

(10) 大的,共享的寄存器文件,可寻址作为:①32 个 32 位的 S(单)寄存器;②32 个 64 位的 D(双)寄存器;③16 个 128 位的 Q(四)寄存器。

3.2.6 性能监视单元

Cortex-A9 处理器包含一个性能监视单元(Performance Monitoring Unit,PMU),该单元提供了 6 个计数器来搜集处理器和存储器系统操作时的统计数字。每个计数器能计数任何在 Cortex-A9 处理器中可用的 58 个事件中的一个。通过来自 CP15 的接口和 DAP 接口,访问 PMU 计数器和它们相关的控制寄存器。

3.3 侦听控制单元

SCU 模块将两个 Cortex-A9 处理器连接到存储器子系统,并且管理两个处理器和 L2 缓存之间的缓存一致性。这个模块负责管理互联仲裁、通信、缓存和系统存储器传输,以及 Cortex-A9 处理器的缓存一致性。APU 也将 SCU 的能力开放给通过 ACP 接口所连接的、PL 内所实现的加速器。这个接口允许 PL 主设备共享和访问处理器的缓存"层次"(不同的缓存结构)。所提供的系统一致性不但改善了性能,也减少了软件的复杂度(否则需要在每个操作系统的驱动程序中负责维护软件的一致性)。

SCU 模块通过一个缓存一致总线(Cache Coherency Bus,CCB)与每个 Cortex-A9 处理器通信,并且负责 L1 和 L2 缓存一致性管理。SCU 支持 MESI 侦听,通过避免不必要的系统访问,改善了功耗和系统性能。模块实现复制 4 路关联标记 RAM,其用作一个本地目录。该目录列出了保存在 CPU L1 数据缓存的一致性缓存行。该目录允许 SCU 高速地检查数据是否是 L1 数据缓存内的数据。并且,不会打断 CPU 的工作。此外,过滤访问,这个访问只能是共享数据的处理器。

SCU 能将一个处理器缓存的干净数据复制到另一个处理器的缓存,这样不需要主存访问来执行这个任务。而且,它能在处理器间移动脏的数据,扫描共享状态和避免写回操作带来的延迟。

注:Cortex-A9 处理器不能直接修改 L1 缓存的内容,因此不保证 L1 指令缓存间的一致性。

3.3.1 地址过滤

SCU 的一个功能是基于它们的地址,过滤处理器和 ACP 产生的交易。并且,将其连接到相应的 OCM 或者 L2 控制器。SCU 内的地址过滤的颗粒度是 1MB。因此,所有用

于处理器的访问或者通过 ACP 的访问，当地址在 1MB 窗口内时，其目标只能是 OCM 或者 L2 控制器。在 SCU 内默认的地址过滤设置是将 4G 地址空间内的高 1MB 和低 1MB 地址连接到 OCM，剩余的地址连接到 L2 控制器。

3.3.2 SCU 主设备端口

每个连接到 L2 或者 OCM 的 SCU AXI 主端口有下面的写和读发布能力：

1）写发布能力

（1）每个处理器 10 个写交易：①8 个非缓存写；②2 个来自 L1 中的替换。

（2）2 个额外的写用于来自 SCU 的替换流量。

（3）来自 ACP 多于 3 个的写交易。

2）读发布能力

每个处理器 14 个读交易：①4 个指令读；②6 个行填充读；③4 个非缓存读；④来自 ACP 多于 7 个读交易。

思考题 3-8：请说明 Cortex-A9 双核处理器的 SCU 的功能和结构特点。

3.4 L2 高速缓存

L2 高速缓存基于 ARM PL310，包含一个 8 路组关联的 512KB 缓存，用于 Cortex-A9 双核处理器。L2 是物理可寻址和物理标记的，支持固定的 32 字节行大小。L2 缓存的主要特点包括：

（1）支持使用 MESI 算法的侦测一致性控制。

（2）提供用于 L2 缓存存储器的奇偶校验。

（3）支持 SMP 模式下的预测读操作。

（4）提供 L1/L2 互斥模式（如数据可以存在其中的一个，但并不是全部）。

（5）每个主设备可以基于主设备、行或者路锁定 L2。

（6）实现 16 个入口深度的预加载引擎，用于加载数据到 L2 缓存存储器。

（7）为改善延迟，支持关键字首行填充。

（8）使用带有决策选项的伪随机替换选择策略：①写通过和写回；②读分配、写分配、读和写分配。

（9）在复位的时候，清除 L2 的数据和标记 RAM 的内容，遵守所要求的安全性。

（10）L2 控制器实现多个 256 位行缓冲区，以改善缓存的效率：①用于外部存储器的行填充缓冲区（Line Fill Buffer，LFB），在 L2 缓存存储器内创建一个完整的缓存行，4 个 LFB 用于支持 AXI 交替读；②两个 256 位缓存行读缓冲区，用于每个从端口，当命中缓存时，这些缓冲区保存来自 L2 缓存的一个缓冲行；③三个 256 位的替换缓冲区保存着来自 L2 缓存淘汰的缓存行，这些缓存行将要写回到主存储器中；④在送到主存储器或者 L2 缓存以前，三个 256 位的保存缓冲区保存着可缓冲的写。这样，可以将到相同缓存行的多个写合并在一起。

（11）在 4k 边界内，控制器实现可选择的缓存行预取操作。

（12）L2 缓存控制器将来自 L1 的互斥请求提交给 DDR、OCM 或者外部存储器。

注：SCU 不能保证在指令和数据 L1 缓存之间的一致性，所以需要使用软件保持其一致性。

L2 高速缓存实现 TrustZone 安全性扩展，用来提供扩展的操作系统安全性。在标记 RAM 上添加一个不安全 NS 的标记位，用来作为一个地址比特位，用于在相同的路内进行查找。NS 标记也添加在所有缓冲区内。标记 RAM 内的 NS 比特位，用来确定到 DDR 和 OCM 的淘汰的安全性。控制器限制非安全访问控制、配置和维护寄存器，限制其对安全数据的访问。

缓存控制器通常的行为取决于 Cortex-A9 的交易。下面给出了不同类型交易的描述。

1）可缓冲的

在到达交易的目的地时，互联或者任何一个它的元件可以将交易延迟任意数量的周期。这通常只和写相关。

2）可缓存的

最终目的的交易不必出现原始交易的特征。对于写，这意味着可以将很多写合并在一起。对于读，这意味着可以预加载一个位置，或者对于多个读只加载一次。确定是否应该缓存一个交易，这个属性应该和读分配和写分配属性一起使用。

3）读分配

对于一个读传输，如果在缓存中缺失，则应该为它分配缓存。如果传输是不可缓存的，则这个属性无效。

4）写分配

对于一个写传输，如果在缓存中缺失，则应该为它分配缓存。如果传输是不可缓存的，则这个属性无效。

在 ARM 的结构中，内在属性用来控制 L1 缓存和写缓冲区的行为，外部的属性输出到 L2 或者外部存储器系统。

类似大多数的现代处理器，在 Cortex-A9 处理器系统中，为了改善性能和功耗，执行很多级的系统优化。这些优化不能完全地隐藏（不被外部看到），这可能引起所希望顺序执行模型的冲突，这些优化例子有：

（1）发出的多个预测和无序执行；

（2）合并加载/保存操作，用于最小化加载/保存延迟；

（3）在一个多核处理器内，基于硬件的一致性管理可能引起缓存行在处理器之间透明的迁移，这样导致不同的处理器核以不同的顺序看到对缓存存储器位置的更新；

（4）当通过 ACP 将外部主设备被包含在一致性系统内时，外部系统特性可以产生新的额外竞争。

因此，定义一个规则是至关重要的，这个规则用于限制一个 CPU 核访问存储器的顺序。这个访问的顺序涉及周围的指令，或者能被多核处理器系统的另一个处理器所观察到。典型地，存储器分成以下几类：

1）标准的

典型的，这种存储器类型应用于所有的数据和可执行的代码。并且，允许预测地读、合并访问和重复地写（当一个异常打断写过程时）。但是，并没有任何不利的影响。当访问标准的存储器时，总是能进行缓冲操作。并且，在大多数条件下，总能缓存这些操作。

但是,它们可以被配置成非缓存的。除了有单纯的地址依赖性和控制依赖性外,访问标准的存储器并不需要有固有的顺序。

2) 强顺序的

典型地,这种存储器应用于存储器映射的外设或者控制寄存器。根据程序所指定的次数访问这些类型的存储器。然而,在存储器访问不同的外设或者在访问不同存储器类型时,并不保证顺序。

3) 设备

这个存储器类型类似于强顺序存储器类型。不同之处在于,设备存储器可以是共享或者是非共享的。

3.4.1 互斥 L2-L1 高速缓存配置

互斥的缓存配置模式中,Cortex-A9 处理器的 L1 数据缓存和 L2 缓存是互斥的。在任何时候,将一个给定的地址缓存在 L1 数据缓存或者 L2 缓存内,但不是所有(也就是说,L1 和 L2 是互斥的)。这样增加了 L2 缓存的可用空间和使用效率。当选择互斥的缓存配置时:

(1) 修改数据缓存替换行策略,这样在 L1 内的淘汰缓存行总是被淘汰到 L2(即使它是干净的);

(2) 如果 L2 缓存的一行是脏的,则来自处理器的对这个地址的读请求,引起写回到外部存储器,以及到处理器的行填充。

必须将所有的 L1 和 L2 缓存行配置成互斥的。通过使用 L2 的辅助控制寄存器的 12 比特位和 Cortex-A9 内的 ACTLR 寄存器的 7 比特,将 L2 和 L1 缓存配置成互斥操作。

对于读,行为如下:

(1) 对于一个命中,缓存行标记为非有效的(复位标记 RAM 的有效位),“脏”位比特不变,如果设置“脏”比特位,未来的访问仍然能命中这个缓存行,但是该行是未来替换的首选;

(2) 对于一个缺失,不分配到该行 L2 缓存。

对于写,取决于 SCU 的属性值。该属性值用来指示写交易是否是一个来自 L1 存储器的替换,以及是否是一个干净的替换。AWUSERS[8]属性表示一个替换,AWUSERS[9]表示一个干净的替换。行为总结如下:

(1) 对于一个命中,该行标记为“脏”,除非 AWUSERS[9:8]=11。这种情况下,“脏”位不变;

(2) 对于一个缺失,如果缓存行被替换(AWUSERS[8]='1'),则分配缓存行。取决于它是否是“脏”的替换,来确定它的“脏位”状态。如果缓存行存在“脏”的替换(AWUSERS[8]='0'),则只有在它被写分配时,才为其分配缓存行。

3.4.2 高速缓存替换策略

辅助控制寄存器的比特[25]用于配置替换策略。该策略是基于轮询或者伪随机的算法。

(1) 轮询替换策略：首先填充无效的和未锁定的路。对于每一行，当路都是有效或者锁定时，替换下来的将选作下一个未锁定的路。

(2) 伪随机替换策略首先填充无效的和未锁定的路；对于每一行，当路都是有效或者锁定时，在未锁定的路之间随机选择替换者。

当要求一个确定的替换策略时，使用锁定寄存器阻止对路进行分配。例如，由于 L2 缓存是 512KB 和 8 路组关联的，每路为 64KB。使用一个确定性的替换策略时，如果一段代码要求驻留在两个路(128KB)，则在代码填充到 L2 缓存前，必须锁定 1～7 路。如果开始的 64KB 只分配给第 0 路，则必须锁定第 0 路，将第 1 路解锁，这样剩下的 64KB 能分配给第 1 路。

这里有两个锁定寄存器，一个用于数据，另一个用于指令。如果要求的话，一个锁定寄存器就能将数据和指令分割到 L2 缓存内各自的路。

3.4.3 高速缓存锁定

允许按照行、路或者主设备(包括 CPU 和 ACP 主设备)，锁定 L2 缓存控制器入口。可以同时实现行锁定和路锁定。然而，路锁定和主设备锁定是互斥的。这是由于路锁定是主设备锁定的子集。

1. 行锁定

当使能时，标记所有新分配的缓存行为锁定状态。控制器认为已经锁定新分配的缓存行，并且不会自然替换这些新分配的缓存行。通过设置行使能寄存器(LER)的[0]比特位，使能行锁定。标记 RAM 的[21]比特位给出了每个缓存行的锁定状态。

注：一个行锁定使能的例子是，当一个关键部分的软件代码加载到 L2 缓存中。

后台对所有的行进行解锁操作，解锁所有由行锁定机制标记为锁定的行。这个操作的状态通过读解锁所有行寄存器(Unlock All Line Register，UALR)进行检查。当正在执行对所有行进行解锁的操作时，用户不能启动一个后台缓存维护操作。如果尝试这样操作，将返回一个 SLVERR 错误。

2. 路锁定

L2 缓存是 8 路组关联的，允许用户将基于路的策略用于锁定替换算法。使能设置计数从 8 路所有的路数减少到直接的映射。32 位的缓存地址由下面的域构成：

```
[Tag Field],[Index Field],[Word Field],[Byte Field]
```

当查找一个缓存时，索引定义了所要查找的缓存路的位置。路数定义了带有相同索引的位置个数，称之为一个组。因此，一个 8 路组关联缓存有 8 个位置。这些位置存在带有索引 A 的地址。在 512K L2 缓存内，有 2^{11} 或者 2024 个标记。

锁定格式 C，正如 ARM 结构参考手册所描述的那样，提供了一种方法，这种方法用于约束替换策略。该策略用于在组内分配缓存行。这种方法使得：

(1) 提取代码或者加载数据到 L2 中；

(2) 保护由于其他访问，导致的替换；

(3) 这个方法也能减少缓存污染。

L2 缓存控制器内的锁定寄存器,用于锁定 L2 缓存内 8 路中的任意一路。为了使用锁定,用户设置每个比特位为 1,用于分别锁定每一路。例如,设置[0]比特用于 0 路,[1]比特用于 1 路。

3. 主设备锁定

主设备锁定特性是路锁定特性的超集。它使能多个主设备共享 L2 缓存,以及使 L2 行为好像是这些主设备有专用较小的 L2 缓存。这个特性使得用户 L2,为特定的主设备 ID 号保留缓存的路。

由主设备实现的 L2 缓存控制锁定,最多只能区分 8 个不同的主设备。然而,在 Cortex-A9 MP 核内最多有 64 个 AXI 主设备 ID 号。表 3.1 给出了将 64 个主 ID 值分组为 8 个可锁定的组。

表 3.1 主设备 ID 组锁定

ID 组	描 述
A9 核 0	来自核 0 的所有 8 个读/写请求
A9 核 1	来自核 1 的所有 8 个读/写请求
A9 核 2	为将来保留
A9 核 3	为将来保留
ACP 组 0	ACP ID={000,111}
ACP 组 1	ACP ID={010,011}
ACP 组 2	ACP ID={100,101}
ACP 组 3	ACP ID={110,111}

3.4.4 使能/禁止 L2 高速缓存控制器

默认情况下,禁止 L2 缓存。通过设置独立于 L1 缓存的 L2 缓存控制寄存器 CCR 的比特 0,使能 L2 缓存。当禁止缓存控制块时,根据它们的地址,传递到 DDR 存储器或者缓存控制器主端口的主互联。由于禁止缓存控制器而导致的地址延迟时间,是来自 SCU 从端口内的一个时钟加上主端口内一个周期的总和。

3.4.5 RAM 访问延迟控制

L2 缓存数据和标记 RAM 使用和 Cortex-A9 处理器一样的时钟。然而,当时钟以它最大的速度运行时,在单周期内访问这些 RAM 是不可行的。为了解决这个问题,L2 缓存控制器提供了一个机制。通过设置标记 RAM 和数据 RAM 延迟控制寄存器(Latency Control Register,LCR)各自的比特[10:8]、[6:4]和[2:0],以调整用于所有 RAM 阵列的写访问、读访问和建立的延迟。对于所有的寄存器,这些位域的默认值是 3b'111,这些默认值对应于用于每个 RAM 阵列三个属性的 8 个 CPU_6x4x 周期的最大延迟。由于这些大的延迟将导致非常低的缓存性能,软件应该按如下复位属性:

(1) 通过写标记 RAM 延迟控制寄存器的比特[10:8]、[6:4]和[2:0]为 3'b001,设置三个标记 RAM 的延迟属性为 2;

(2) 通过写数据 RAM 延迟控制寄存器的比特[10:8]和[2:0]为 3'b001,设置数据 RAM 的写访问和建立延迟属性为 2;

(3) 通过写数据 RAM 延迟控制寄存器的比特[6:4]为 3'b010,设置数据 RAM 的读访问延迟属性为 3。

3.4.6 保存缓冲区操作

如果在第一个访问后,控制器没有耗尽保存缓冲区,而出现两个到相同地址和相同安全比特位的缓冲写访问,则将覆盖第一个写访问。保存缓冲区具有合并操作的能力,这样它将对一个相同行地址连续的写操作合并到相同的缓冲区槽。这意味着只要它们包含数据,控制器就不会耗尽槽。而是,等待目标是相同缓存行的其他潜在的访问。保存缓冲区的耗尽策略如下(从端口是指从 SCU 到 L2 缓存控制器的端口):

(1) 如果目标是设备存储器区域,保存缓冲区槽被立即耗尽;

(2) 一旦它们满,则耗尽保存缓冲区槽;

(3) 在从端口发生每一个强顺序读时,耗尽缓冲区;

(4) 在从端口发生每一个强顺序写时,耗尽缓冲区;

(5) 如果保存缓冲区包含数据,耗尽最近访问的槽;

(6) 如果一个保存缓冲区槽检测到有风险时,则耗尽缓冲区,用于解决风险,当数据出现在缓存缓冲区时,发生风险,但是出现在缓存 RAM 或者外部存储器时,不会出现风险;

(7) 当从端口接收到一个锁定的交易时,耗尽保存缓冲区;

(8) 当从端口接收到一个目标是配置寄存器的交易时,耗尽保存缓冲区。

合并条件基于地址和安全属性。数据在保存缓冲区,并且没有耗尽时,发生合并操作。当一个写分配可缓存的槽被耗尽时,缓存缺失,并且没有满。保存缓冲区通过主端口发送一个请求到主互联或者 DDR,以完成缓存行。相应的主端口通过互联发送一个读请求,并且提供数据到保存缓冲区。当槽满时,它能分配到缓存行。

3.4.7 在 Cortex-A9 和 L2 控制器之间的优化

为了提高性能,SCU 和 L2 控制器存在接口。并且,有一部分和 OCM 进行连接,用于实现下面的优化:

(1) 早期的写响应;

(2) 预取提示;

(3) 充分的行写零填充;

(4) Cortex-A9 多核处理器预测地读。

这些优化应用到来自处理器的传输,不包括 ACP。

1. 早期的写响应

来自 Cortex-A9 到 L2 缓存控制器的写交易期间，只有当最后一拍数据到达 L2 控制器时，来自 L2 控制器的写响应才正常返回到 SCU。这个优化使得只要保存缓冲区接受写地址，并且允许 Cortex-A9 处理器能为写提供更高的带宽时，L2 控制器就发送某个写交易的写响应。默认时，禁止这个特性。用户能通过设置用于 L2 控制器的 ACR 的早期 BRESP 使能位来使能这个特性。Cortex-A9 不要求通过任何编程来使能这个特性。OCM 不支持这个特性，正常地产生它的写响应。

2. 预取提示

当配置 Cortex-A9 为 SMP 模式时，在 CPU 内自动实现数据预取，向 L2 缓存控制器发送读访问。这些特殊的读称为预取提示。当 L2 控制器接收到这个预取提示时，它为一个缺失分配目标缓存行到 L2 缓存。但是，并不返回任何数据到 Cortex-A9 处理器。用户能通过下面两个方法，使能 Cortex-A9 处理器预取提示生成：

(1) 通过设置 ACTLR 寄存器的比特[1]，使能 L2 预取提示特性。当使能时，当它检测到一个一致性存储器上的规则取模式时，这个特性设置 Cortex-A9 处理器自动发布 L2 预取提示请求。

(2) 使用预加载引擎(Pre-Load Engine，PLE)。当在 Cortex-A9 内使用这个特性时，PLE 在编程的地址上发送一系列的 L2 预取提示请求。

不要求对 L2 控制器的额外编程。将预取提示应用到 OCM 存储器空间时，不引起任何的行为。这是因为不像缓存那样，在将数据传输到 OCM RAM 时，要求软件明确的操作。

3. 充分的行写零

当使能这个特性时，在单个写命令周期内，Cortex-A9 处理器能用 0，将整个非一致性的缓存行写到 L2 缓存。这提供了性能改善和功耗的降低。当一个 CPU 正在执行存储器分配 memset 例程来初始化一个特殊的存储器区域时，Cortex-A9 很可能使用这个特性。

默认时，禁止该特性。通过设置用于 L2 控制器的辅助控制寄存器的充分的行写零使能比特位和 Cortex-A9 ACTLR 寄存器的使能位，使能该特性。使能该特性，必须执行下面的步骤：

(1) 在 L2 内使能充分的行写零特性；

(2) 使能 L2 缓存控制器；

(3) 在 A9 内使能充分的行写零特性。

缓存控制器不支持带有这个特性的强顺序写访问。如果在 Cortex-A9 内使能这个特性，OCM 也支持这个特性。

4. Cortex-A9 预测的读

这个特性是 Cortex-A9 多核处理器独一无二的配置。通过使用 SCU 控制寄存器的一个专门的软件控制比特位，使能这个特性。对于这个特性，通过使用 ACTLR 寄存器

内的 SMP 比特位，使 Cortex-A9 处于 SMP 模式。然而，L2 控制器不要求任何指定的设置。当使能预测的读特性时，在一致性行填充上，SCU 将带有它标记查找表的读交易预测的发布到控制器。在这些预测的读操作时，控制器并不返回数据，而只在它的行读缓冲区内准备数据。

如果 SCU 缺失，它发布一个确认行填充到控制器。确认和控制器内前面的预测读合并，使得控制器返回数据到 L1 缓存控制器，比 L2 缓存命中快。如果 SCU 命中，在某个周期后或者当存在资源冲突时，L2 自然地终止预测读。当预测读结束时，L2 控制器通知 SCU，或者确认或者终止。

3.4.8 预取操作

预取操作使得能够事先从存储器取缓存行，用于提高系统性能。用户通过设置预取控制寄存器的 28 或者 29 比特位，使能预取特性。当使能时，如果来自 SCU 的从端口接收到一个可缓存的读交易时，在随后的缓存行执行一个缓存查找。预取控制寄存器的[4:0]比特提供了随后缓存行的地址。如果发生缺失时，从外部存储器取出缓存行，并且分配到 L2 缓存。

默认时，预取偏置是 5'b00000。例如，如果 S0 接收到在地址 0x100 上的一个可缓存的读，预取 0x120 地址的缓存行。预取下一个缓存行可能不产生性能优化。在一些系统中，超前预取可以达到更好的性能。通过将预取缓存行的地址设置到缓存行＋1＋偏置的地址，使得预取偏置能够达到更好的特性。预取偏置的优化值取决于外部存储器读延迟和 L1 读发布能力。预取机制不能跨越 4KB 的边界。

在控制主端口内，预取访问能使用大量的地址槽。这可以防止对非预加载访问的服务，它将影响性能。为了应对这个影响，控制器放弃预取访问。通过使用预取控制寄存器的 24 比特位来控制它。当使能时，如果在控制器主端口内的预取和非预取之间存在资源冲突，则放弃预取访问。当所放弃的这些预取访问数据从外部存储器返回时，放弃它，并且不把它分配给 L2 高速缓存。

3.4.9 编程模型

下面应用于 L2 缓存控制器内的寄存器：

(1) 通过存储器映射的寄存器集合控制缓存控制器。这些寄存器的存储器区域必须在 L1 页表内，定义为强顺序或者设备存储器属性。

(2) 必须保护所有寄存器的保留位；否则，可能发生设备未定义的行为。

(3) 除非在相关的文字中进行了描述，所有寄存器支持读和写访问。写操作用于更新寄存器的内容，读操作返回寄存器的内容。

(4) 在处理所有对寄存器的写操作前，自动执行一个初始的同步操作。

作为一个例子，由下列寄存器操作构成一个典型的缓存控制器启动编程序列。

(1) 使用一个读修改写，来设置全局配置。写到辅助、标记 RAM 延迟、数据 RAM 延迟、预取和电源控制寄存器：①关联性和路大小；②用于 RAM 访问的延迟；③分配策略；④预取和供电能力。

(2) 通过路的安全写来无效,偏移为 0x77c,使得缓存的所有入口无效:①写 0xFFFF 到 0x77c;②轮询缓存维护寄存器,直到完成使无效的操作。

(3) 如果要求,写寄存器 9 锁定数据 D 和指令 I。

(4) 写到中断清除寄存器。清除任何所剩余的、原来的中断设置。

(5) 如果希望使能中断,则写到中断屏蔽寄存器。

(6) 使用最低有效位写控制器 1,将其设置为 1,来使能缓存。

使能 L2 缓存时,如果执行一个到辅助、标记 RAM 延迟或者数据 RAM 延迟控制寄存器的写操作,则导致 SLVERR 错误。在写这些寄存器前,通过写控制寄存器来禁止 L2 缓存。

思考题 3-9:请说明 Cortex-A9 双核处理器内 L2 缓存的结构特点和功能。

思考题 3-10:请参阅相关书籍,说明缓存中的组、路和行的含义,以及它们之间的关系。

3.5 片上存储器

片上存储器(On-Chip Memory,OCM)模块,包括 256KB 的 RAM 和 128KB 的 ROM (BootROM)。

3.5.1 片上存储器结构

片上存储器模块,支持两个 64 位 AXI 从接口端口,其中一个用于 CPU/ACP 通过 APU 内的 SCU 进行访问;另一个由 PS 和 PL 内的所有其他总线主设备进行共享。BootROM 空间专用于启动过程,对用户来说是不可见的,即用户不需要关心这个启动的过程。OCM 支持高吞吐量的 AXI 读和写操作,用于 RAM 访问(RAM 是 1 个单端口双宽度 128 位的存储器)。为了充分利用 RAM 较高访问吞吐量,用户应用程序必须使用偶数个 AXI 猝发大小和 128 位对齐的地址。

TrustZone 特性支持 4KB 存储器粒度。整个 256K RAM 能分成 64 个 4KB 的存储块,并且单独地分配安全属性。

如图 3.5 所示,OCM 上有 10 个相关的 AXI 通道,5 个用于 CPU/ACP(SCU)端口,5 个用于其他 PS/PL 主设备(OCM 开关端口)。在 OCM 模块内,执行 SCU 和 OCM 开关端口的读和写通道之间的仲裁。只在访问 RAM 时,产生奇偶校验并进行检查。其他主接口有一个中断信号 Irq,以及一个寄存器用于访问 APB 端口。

OCM 模块的关键特性包括:

(1) 片上 256KB RAM;

(2) 片上 128KB BootROM;

(3) 两个 AXI 3.0 的 64 位从接口;

(4) 用于 CPU/ACP 读 OCM 的低延迟路径(CPU 在 667MHz,最小 23 个周期);

(5) 在 OCM 互联端口(非 CPU 端口)上的读和写 AXI 通道之间,采用轮询预仲裁;

(6) 在 CPU/ACP(通过 SCU),以及 OCM 互联 AXI 端口之间,采用固定优先级仲裁;

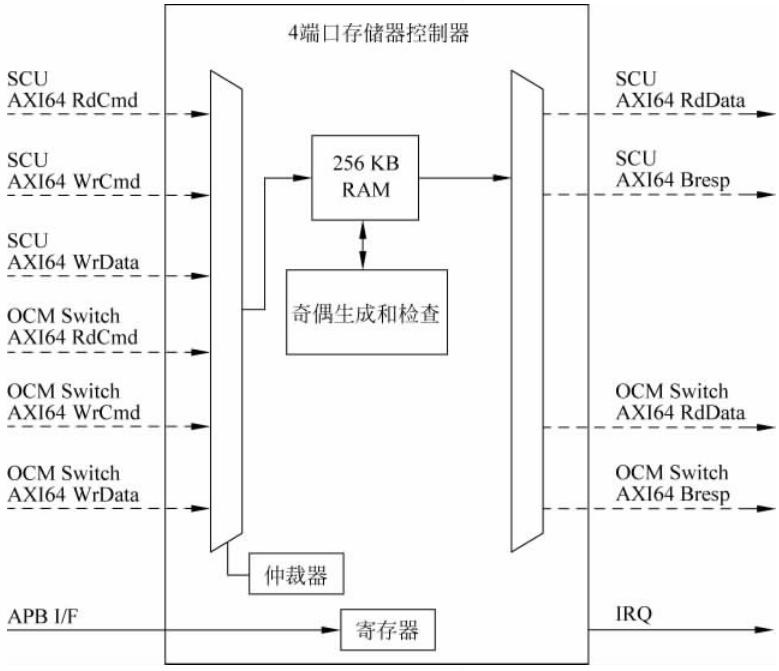

图 3.5　OCM 块图

(7) 在 OCM 互联端口上,支持同时读和写命令所需要的 AXI 64 位充分带宽(带有优化的对齐限制);

(8) 支持来自 AXI 主设备的随机访问;

(9) TrustZone 支持带有 4KB 页颗粒度的 RAM;

(10) 灵活的地址映射能力;

(11) 支持 RAM 按照字节产生奇偶校验,检查和中断;

(12) 支持在 CPU(SCU)端口上,下面非 AXI 的特性:①零行填充;②预取提示;③早期 BRESP;④预测的行预取。

图 3.6 给出了从系统的角度描述的 OCM 系统结构。

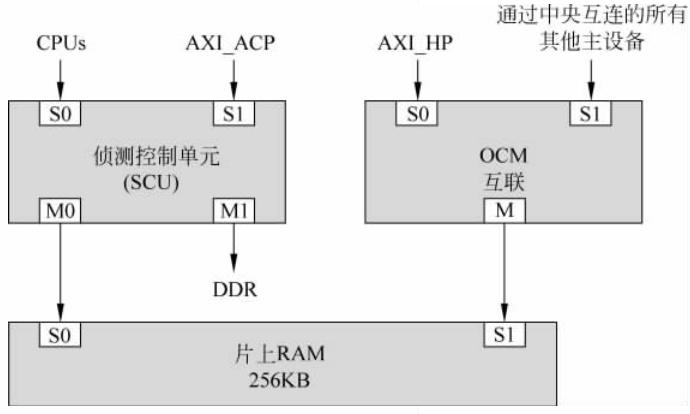

图 3.6　OCM 系统

3.5.2 片上存储器功能

OCM模块主要由RAM存储器模块组成。OCM模块也包含仲裁、成帧、奇偶校验和中断逻辑。

1. 优化的传输对齐

RAM是一个单端口的,双宽度(128比特)模块。它可以在一些指定的条件下,模拟一个双端口存储器。当使用128位对齐时,自动模拟产生双端口操作,甚至使用多个AXI猝发命令访问64位宽度的OCM AXI接口。对猝发进行优化,理论上可以达到100%的RAM吞吐量。如果猝发没有对齐128位,或者猝发宽度是多个奇数个64位时,模块内的控制逻辑自动重新对齐传输,给出RAM起始和结束地址,作为64位的操作,而不是用更优化的128位操作。

不推荐下面的操作:

(1) 配置OCM存储器作为MMU内的设备存储器;

(2) 通过ACP端口时,使用窄的、不可修改的访问。

这是因为这种类型的流量模式不能利用双宽度存储器的优势,使得OCM的效率降低到25%。

2. 时钟

OCM模块由CPU_6x4x时钟驱动。然而,RAM本身是个例外,它由CPU_2x时钟驱动。尽管它的128位宽度是任何进入64位宽度AXI通道的一倍。OCM开关送到OCM模块,它由CPU_2x时钟驱动。SCU由CPU_6x4x时钟驱动。

3. 仲裁策略

除了CPU和ACP外,假设所有AXI总线的主设备没有强延迟要求。因此,在两个AXI从设备接口之间,OCM使用固定的仲裁策略(基于数据拍)。默认的优先级递减顺序是:

(1) SCU-读;

(2) SCU-写;

(3) OCM-切换。

使用ocm.OCM_CONTROL.ScuWrPriorityLo寄存器设置,递减的优先级仲裁可以修改为:

(1) SCU-读;

(2) SCU-切换;

(3) OCM-写。

仲裁按照图3.7实现。

这儿有一个额外的轮询预仲裁过程,使得每个数据拍可以在一个读或者写交易之间进行选择,用于OCM开关端口流量。

注:(1) 执行仲裁是基于传输(数据拍或者时钟周期),而不是基于AXI命令;在仲

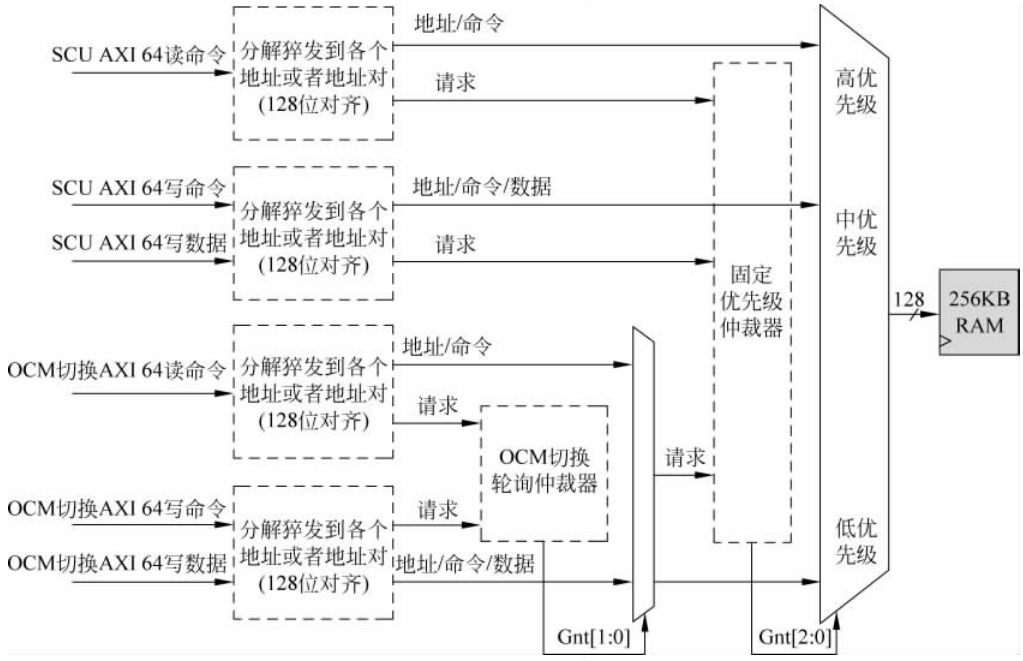

图 3.7　默认(ScuWrPriorityLo=0)OCM 仲裁

裁前,进入的 AXI 读和写命令被分割成各个地址(或者 128 位地址对,用于对齐猝发操作)。

(2) 每个单个的写地址拍不要求访问存储器阵列,直到和它相关的写数据在 OCM 模块内可用为止,这将防止由于没有可用的数据,而停止一个写请求。

对于 OCM 上的系统限制是:

(1) RAM 阵列和 OCM 开关端口由 CPU_2x 驱动,其工作频率为 CPU 时钟的 1/2 或者 1/3;

(2) SCU(CPU/ACP)端口由全速率 CPU 时钟驱动;

(3) 4 个进入的 AXI 数据通道的每一个通道都是 64 位宽度;

(4) RAM 阵列为 128 位宽度;

(5) OCM 开关端口有单独的读和写通道,这些通道可以同时活动;

(6) SCU(CPU/ACP)端口有单独的读和写通道,这些通道可以同时活动;

(7) SCU(CPU/ACP)端口有固定的仲裁优先级,默认其优先级高于 OCM 开关端口。

由于这些限制的结果,在 SCU CPU 或者 ACP 接口上产生 RAM“饱和”,而“饿死”OCM 开关和它所服务的主设备。然而,如果 CPU 运行时,打开高速缓存。在这个速度下,它们产生到这个模块新的命令是非常少的。这样允许 OCM 开关端口共享 RAM。也可以提升 OCM 开关的仲裁优先级,使其高于 SCU 写通道的优先级。

4. 地址映射

可以修改分配给 OCM 模块的地址访问,使其存在于地址映射空间最开始和最后的 256KB 上,这样可以灵活地管理 ARM 低或者高异常向量模式。此外,使用 SCU 地址过

滤功能，CPU 和 ACP AXI 接口能使它们最低的 1MB 地址范围的访问转移到 DDR。

当寻址 OCM 时，必须考虑下面的细节：

(1) OCM 模块响应地址范围为(0x0000_0000～0x0007_FFFF)，访问 RAM 阵列没有映射的地址，将给出一个错误的响应；

(2) 通过 4 位的 slcr.OCM_CFG[RAM_HI]，以 4 个独立的 64KB 区域的粒度，将 256KB RAM 阵列映射到低地址范围(0x0000_0000～0x0003_FFFF)，或者高地址范围(0xFFFC_0000～0xFFFF_FFFF)；

(3) 在任何形式的复位时由硬件设置 SCU 地址过滤域：

```
mpcore.SCU_CONTROL_REGISTER[Address_filtering_ enable]
```

(4) 用户不应该禁止，这对非 OCM 的地址过滤是必要的，这是为了在两个下游 SCU 端口之间正确地连接交易，地址过滤范围为 1MB 的颗粒度；

(5) SCU 地址过滤特性，将来自 CPU 和 ACP 主设备所访问的目标地址范围(0x0000_0000～0x000F_FFFF)(其地址包含了 OCM 的低地址范围)，重新映射到 PS DDR DRAM，其不依赖 RAM 地址的设置；

(6) 其他所有没有穿过 SCU 的主设备，总是不能访问 OCM 低地址范围(0x0000_0000～0x0007_FFFF)内的较低的 512KB DDR 空间。

在进入用户模式后，不能再访问 BootROM，并且与 RAM 空间分开。注意一个 64KB 范围驻留在 OCM 高地址范围。其他 192KB 驻留在低地址范围。表 3.2 和表 3.3 给出了初始的 OCM/DDR 地址映射和寄存器设置。

表 3.2 初始的 OCM/DDR 地址映射

地址范围(十六进制)	大小	CPU/ACP	其他主设备
0000_0000～0000_FFFF	64KB	OCM	OCM
0001_0000～0001_FFFF	64KB	OCM	OCM
0002_0000～0002_FFFF	64KB	OCM	OCM
0003_0000～0003_FFFF	64KB	保留	保留
0004_0000～0007_FFFF	256KB	保留	保留
0008_0000～000F_FFFF	512KB	保留	DDR
0010_0000～3FFF_FFFF	1023MB	DDR	DDR
FFFC_0000～FFFC_FFFF	64KB	保留	保留
FFFD_0000～FFFD_FFFF	64KB	保留	保留
FFFE_0000～FFFE_FFFF	64KB	保留	保留
FFFF_0000～FFFF_FFFF	64KB	OCM	OCM

表 3.3 初始的寄存器设置

寄 存 器	值
slcr.OCM_CFG[RAM_HI]	1000
mpcore.SCU_CONTROL_REGISTER[Address_filtering_enable]	1
mpcore.Filtering_Start_Address_Register	0x0010_0000
mpcore.Filtering_End_Address_Register	0xFFE0_0000

对于一个连续的 RAM 地址映射，通过 SLCR 寄存器将 0x0000_0000～0x0002_FFFF 地址范围的 RAM 重新定位到基地址 0xFFFF_C0000。

slcr.OCM_CFG[RAM_HI]的每一位对应到一个 64 位的范围，MSB 对应到最高的地址偏置范围。

表 3.4 和表 3.5 给出了一个 OCM 重定位地址映射和 OCM 重定位寄存器的设置。

表 3.4　OCM 重定位的地址映射的例子

地址范围(十六进制)	大小	CPU/ACP	其他主设备
0000_0000～0000_FFFF	64KB	保留	保留
0001_0000～0001_FFFF	64KB	保留	保留
0002_0000～0002_FFFF	64KB	保留	保留
0003_0000～0003_FFFF	64KB	保留	保留
0004_0000～0007_FFFF	256KB	保留	保留
0008_0000～000F_FFFF	512KB	保留	DDR
0010_0000～3FFF_FFFF	1023MB	DDR	DDR
FFFC_0000～FFFC_FFFF	64KB	OCM	OCM
FFFD_0000～FFFD_FFFF	64KB	OCM	OCM
FFFE_0000～FFFE_FFFF	64KB	OCM	OCM
FFFF_0000～FFFF_FFFF	64KB	OCM	OCM

表 3.5　OCM 重定位寄存器设置

寄　存　器	值
slcr.OCM_CFG[RAM_HI]	1111
mpcore.SCU_CONTROL_REGISTER[Address_filtering_enable]	1
mpcore.Filtering_Start_Address_Register	0x0010_0000
mpcore.Filtering_End_Address_Register	0xFFE0_0000

CPU 和 ACP 通过 SCU 端口所看到 OCM 的视图和通过 OCM 开关所看到的其他主设备是不同的。不同于 OCM，SCU 用它自己专门的地址过滤机制寻址从设备。而系统内的其他总线主设备，则通过内建在系统互连内的固定地址译码机制进行连接。

这些其他总线主设备通过访问该地址范围，即地址 0x0000_0000～0x0007_FFFF 和地址 0xFFFF_C000～0xFFFF_FFFF，而进入到 OCM 空间。通过访问这些地址范围，其他总线主设备总是能“看到”OCM。取决于对 SLCR OCM 寄存器的配置，当这些访问终止于 RAM 阵列或者一个默认的保留地址时，将导致 AXI SLVERR 错误。在 RAM 地址映射空间内，这些其他主设备潜在地看到“间隙”。

然而，使用 SCU 地址过滤，CPU/ACP 视图可能是不同的。例如，如果 CPU 想让 DDR DRAM 位于地址 0x0000_0000，它能配置地址过滤和 SLCR OCM 寄存器，这样就能看到表 3.6 所给出的地址映射范围。在表 3.6 中，注意，CPU/ACP 主设备能寻址整个的 DDR 地址范围，而其他主设备不能寻址 DDR 较低的 512KB 空间。表 3.7 给出了 OCM 重定位寄存器的设置。

表 3.6 OCM 重定位的地址映射的例子

地址范围(十六进制)	大 小	CPU/ACP	其他主设备
0000_0000～0000_FFFF	64KB	DDR	保留
0001_0000～0001_FFFF	64KB	DDR	保留
0002_0000～0002_FFFF	64KB	DDR	保留
0003_0000～0003_FFFF	64KB	DDR	保留
0004_0000～0007_FFFF	256KB	DDR	保留
0008_0000～000F_FFFF	512KB	DDR	DDR
0010_0000～3FFF_FFFF	1023MB	DDR	DDR
FFFC_0000～FFFC_FFFF	64KB	OCM	OCM
FFFD_0000～FFFD_FFFF	64KB	OCM	OCM
FFFE_0000～FFFE_FFFF	64KB	OCM	OCM
FFFF_0000～FFFF_FFFF	64KB	OCM	OCM

表 3.7 OCM 重定位寄存器设置

寄 存 器	值
slcr. OCM_CFG[RAM_HI]	1111
mpcore. SCU_CONTROL_REGISTER[Address_filtering_enable]	1
mpcore. Filtering_Start_Address_Register	0x0000_0000
mpcore. Filtering_End_Address_Register	0xFFE0_0000

5. 中断

在下面的环境下,OCM 模块能确定到 APU 的一个中断:

(1) 单比特奇偶错误;

(2) 多比特奇偶错误;

(3) 不支持的 LOCK 请求。

通过设置 OCM. OCM_PARITY_CTRL 寄存器,使能所有的中断。通过访问 OCM. OCM_IRQ_STS 寄存器,访问各个中断状态。并且,通过给相应的位置写 1 清除中断。

当没有确认 OCM. OCM_PARITY_CTRL[ParityCheckDis]时,在 RAM 阵列上执行奇偶校验。当使能奇偶校验时,一个单比特或者多比特奇偶错误将设置合适的中断状态。并且,如果设置了相关的使能位,将触发一个外部中断。第一个奇偶错误的地址偏移保存在 OCM. OCM_PARITY_ERRADDRESS 寄存器中。对于读,能给请求的主设备发布一个 SLVERR 响应,它用于那些不能或者不喜欢处理中断的设备。

6. 编程模型

表 3.8 给出和 OCM 相关的部分寄存器列表。

下面给出重新组织 OCM 地址空间和执行 DDR 重映射的步骤:

(1) 通过发布数据(DSB)和指令(ISB)同步屏障指令,完成所有提交的交易。

(2) 由于地址映射的改变,可能阻止取出旁边的指令;使能 L1 指令缓存和预取缓存行,用于剩余的功能。典型地,使用 PL1 指令。

(3) 为了方便预取,考虑将预取的指令对齐到缓存行边界的开始。

(4) 通过发布 ISB 指令,确认完成预取指令。

(5) 通过写解锁关键值到 slcr. SLCR_unlock 寄存器,来解锁 SLCR。

(6) 修改 slcr. OCM_CFG 寄存器来改变地址的范围,用于 RAM 响应。

(7) 如果期望的话,通过写锁定关键值到 slcr. SLCR_lock 寄存器,重新锁定 SLCR。

(8) 修改 Filtering_Start_Address_Register 寄存器内所期望交易的开始地址,这些交易来自 OCM 或者 SCU 的主设备,应该将其过滤。典型地,设置为 0x0010_0000(默认的,不要重定位较低的 1MB),0x0000_0000(开始重定位最低的地址到 DDR RAM)。

(9) 修改 Filtering_End_Address_Register 寄存器内所希望交易的结束地址,这些交易来自 OCM 或者 SCU 的主设备,应该被过滤掉。一个典型的设置为 0xFFE0_0000。

(10) 设置 mpcore. SCU_CONTROL_REGISTER[Address_filtering_enable]寄存器,使能地址过滤。

(11) 通过发布一个数据存储器屏障(Data Memory Barrier,DMB)指令,确认到 SLCR 的访问已经完成。这允许随后的访问依赖于新地址映射。

表 3.8 OCM 寄存器概述

模块	寄存器名字	概 述
OCM	OCM_PARITY_ERRADDRESS	返回 RAM 奇偶错误的地址
	OCM_PARITY_CTRL	设置中断使能,AXI 读响应错误使能,奇偶校验使能,奇校验生成
	OCM_IRQ_STS	读中断状态,清除中断
	OCM_CONTROL	改变预仲裁优先级
slcr	SLCR_LOCK	SLCR 寄存器写禁止
	SLCR_UNLOCK	SLCR 寄存器写使能
	OCM_RST_CTRL	OCM 子系统复位
	TZ_OCM_RAM0/1	OCM TrustZone
	OCM_CFG	配置 RAM 地址映射
mpcore	SCU_CONTROL_REGISTER	SCU 地址过滤使能
	Filtering_Start_Address_Register	SCU 地址过滤基地址
	Filtering_End_Address_Register	SCU 地址过滤结束地址

思考题 3-11:请说明 OCM 的结构特点和实现的功能。

3.6 APU 接口

APU 接口主要包括 PL 协处理器接口和中断接口两个部分。

3.6.1 PL 协处理接口

1. ACP 接口

加速器一致性端口 ACP 是 SCU 上的一个 64 位的 AXI 从接口,提供了来自 PL 到 Cortex-A9 MP 核处理器子系统的异步的缓存一致性访问点。系统的 PL 主设备可以使

用这个接口，就类似 APU 一样能正确地访问缓存和存储器子系统。这样就简化了软件，提高了整个系统的性能或者改善了功耗。这个接口作为一个标准的 AXI 从端口，支持所有的标准读和写交易，而不需要在 PL 元件上放置额外的一致性要求。因此，ACP 提供了来自 PL 到 ARM 缓存的缓存一致性的访问，而任何到 PL 的本地存储器并不和 ARM 保持一致性。

任何通过 ACP 到存储器一致性区域的读交易和 SCU 进行交互，用于检查所要求的信息是否保存在处理器 L1 缓存内。如果是这样，数据直接返回到正在请求的元件。如果在 L1 缓存内缺失，在最终提交到主存以前，仍然有机会命中 L2 缓存。对于到任何一致性存储器区域的写交易，在写提交到存储器系统前，SCU 强迫一致性。交易也可选择分配到 L2 缓存，用于消除对片外存储器的写通过操作，这可能对系统功耗和性能产生影响。

2. ACP 请求

在 ACP 上执行读和写请求行为是不同的，这取决于请求是否是一致性的。这个行为如下：

1）ACP 一致性读请求

当 ARUSER[0]=1，ARCACHE[1]=1 和 ARVALID 时，一个 ACP 读请求是一致性的。在这种情况下，SCU 强迫一致性。当数据出现在其中的一个 Cortex-A9 处理器时，直接从相关的处理器读取数据，并且返回到 ACP 端口。当数据没有出现在其中的一个 Cortex-A9 处理器时，在 SCU AXI 其中的一个主端口上的发布读请求，以及除了锁定属性外的所有参数。

2）ACP 非一致性读请求

当 ARUSER[0]=0 或者 ARCACHE[1]=0 和 ARVALID 时，一个 ACP 读请求是非一致性的。在这种情况下，SCU 不强迫一致性。读请求直接提交 SCU AXI 主设备上可用的一个端口上，这个端口到 L2 缓存控制器或者 OCM。

3）ACP 一致性写请求

当 AWUSER[0]=0 或者 AWCACHE[1]=0 和 AWVALID 时，一个 ACP 写请求是一致性的。在这种情况下，SCU 强迫一致性。当数据出现在其中的一个 Cortex-A9 处理器中时，首先清除来自该 CPU 数据，并使其无效。当数据没有出现在其中的一个 Cortex-A9 处理器中时，或者已经被清除和无效时，在一个 SCU AXI 主端口上发布写请求，以及除了锁定属性外所有相应的 AXI 参数。

注：*如果设置了相应的写参数，则交易可选择分配到 L2 缓存。*

4）ACP 非一致性写请求

当 AWUSER[0]=1 或者 AWCACHE[1]=0 和 AWVALID 时，一个 ACP 写请求是非一致性。在这种情况下，SCU 不强迫一致性，写请求直接提交给一个可用的 SCU AXI 主端口。

3. ACP 用法

与传统缓存刷新和加载策略相比，ACP 为 PS 和 PL 内实现的加速器之间提供了低延迟通道。下面给出了基于 PL 实现加速器的例子：

(1) 在它的本地缓存空间内,CPU 为加速器准备输入数据;

(2) CPU 使用到 PL 的通用 AXI 主接口,将消息发送到加速器;

(3) 加速器通过 ACP 取数据,处理数据,然后通过 ACP 返回结果;

(4) 通过写到一个已知的位置,加速器设置一个标志,用来表示正在处理的数据完成;可以通过处理器轮询这个标志的状态或者产生一个中断。

表 3.9 给出了基于当前缓存状态的 ACP 读和写行为。很清楚,当发生命中缓存时,有很小的延迟。

表 3.9 ACP 读和写行为

行 为	描 述
ACP 读-I(无效)	SCU 通过两个 AXI 主接口中的一个从外部存储器中取数,数据直接提交给 ACP,它不影响 CPU L1 缓存状态
ACP 读-M(修改)	SCU 从 L1 缓存取出带有 M 状态的数据,它不影响 L1 缓存的状态
ACP 读-S(共享)	SCU 从 L1 缓存取出带有 S 状态的数据,它不影响 L1 缓存的状态
ACP 读-E(独占)	SCU 从 L1 缓存取出带有 E 状态的数据,它不影响 L1 缓存的状态
ACP 写-I(无效)	通过两个 AXI 主接口中的一个将数据写到外部存储器中,它不影响 CPU L1 缓存状态
ACP 写-M(修改)	在 L1 缓存中带有 M 状态的数据,首先被刷新到外部存储器;之后,ACP 数据写入外部存储器接口,先前带有 M 状态的 L1 缓存变成 I 状态;如果 SCU 写覆盖整个的缓存行,则跳过 L1 缓存刷新
ACP 写-S(共享)	通过两个 AXI 主接口中的一个将数据写到外部存储器中,先前带有 S 状态的 L1 缓存变成 I 状态
ACP 写-E(独占)	通过两个 AXI 主接口中的一个将数据写到外部存储器中,任何先前带有 S 状态的 L1 缓存变成 I 状态

4. ACP 限制

ACP 端口有如下限制:

(1) 一致性存储器不允许互斥访问;

(2) 一致性存储器不允许锁定访问;

(3) 长度=3,大小=3 和写选通≠11111111 的写交易,可能破坏 CPU 内的缓存行;

(4) ACP 连续地访问 OCM,能"饿死"来自其他 AXI 主设备的访问。为了允许来自其他主设备的访问,ACP 到 OCM 的带宽应该调整为小于 OCM 的峰值带宽。通过将猝发大小调整为小于 8 个 64 比特的字,实现对带宽的调整。

思考题 3-12:Cortex-A9 双核处理器的 ACP 端口的结构特点和功能。

5. 事件接口

事件总线提供了低延迟和直接的机制,在 APU 和 PL 之间传输状态和实现唤醒机制。在这个接口上,事件的输入和输出信号使用切换发出信号,通过在所有的边沿将信号切换到相反的逻辑电平,与一个事件通信。事件总线包括这些信号:

1) EVENTEVENTO

一个切换输出信号,表示其中的一个 CPU 正在执行 SEV 指令。

2) EVENTEVENTI

一个切换输入信号，它将唤起其中一个或者所有的由于执行 WFE 指令而进入待机状态的 CPU。

3) EVENTSTANDBYWFE[1:0]

两级输出信号表示两个 CPU 的状态，如果相应的 CPU 在执行 WFE(等待事件)指令后，进入待机状态，则确认该比特位。

4) EVENTSTANDBYWFI[1:0]

两级输出信号表示两个 CPU 的状态，如果相应的 CPU 在执行 WFI(等待中断)指令后，进入待机状态，则确认该比特位。

事件总线能用于实现基于 PL 的加速器。事件输出能用于触发一个 ACP 加速器，从预定义的地址进行读操作。更进一步，在处理中，输入能用于通信，数据经过 ACP 已经写回，准备被 CPU“消费”。这个例子的详细描述如下：

(1) CPU0 产生数据，L1/L2 缓存内的加速器要求这个数据，这个数据包含需要被处理的命令和信息；

(2) CPU0 发布 SEV(发送事件)指令，使得 EVENTEVENTO 切换到 PL，信号连接到 PL 内实现的一个加速器 IP；

(3) CPU0 发布一个 WFE(等待事件)指令，将 CPU 设置在低功耗待机状态，通过输出到 PL 的 EVENTSTANDBYW[0]状态位表现；

(4) 加速器注意切换的 EVENTEVENTO 信号，识别 CPU0 正在等待，加速器通过 ACP 接口从预先安排的地址和数据格式中取出数据，并且开始处理；

(5) 当写结果数据通过 ACP 返回时，加速器确认 EVENTENENT1 输入，用来表示完成处理，并且唤醒 CPU0；

(6) 从待机状态唤醒 CPU0，它在 EVENTSTANDBYWFE[0]的输出中表现，CPU0 使用处理过的数据继续执行。

3.6.2 中断接口

PS 通用中断控制器(General Interrupt Controller，GIC)支持 64 个中断输入线，这些中断线由 PS 或者 PL 内的其他模块驱动。64 个中断中的其中 6 个由 APU 驱动。这些包括 L1 奇偶校验失败，L2 中断(所有原因)和性能监视器单元(Performance Monitor Unit，PMU)中断。GIC 的中断输出驱动每个 Cortex-A9 处理器的 IRQ 或者 FIQ 输入。通过 APU 内的 SCU 寄存器，选择中断哪个处理器。

3.7 APU 内的 TrustZone

如图 3.8 所示，对于同时运行带有安全和不安全的应用程序，系统从上电复位状态转移到稳定状态需要下面的过程。

图 3.8 中，假设打开设备安全性。然而，没有必要要求使能 TrustZone 安全性。在该图 3.8 中，实线用来显示启动流程，虚线用来说明系统运行后的处理过程。阴影的模块是软件功能块。在系统启动引导后，保持运行。在 TrustZone 启动流程中，首先启动安全 OS。用于初始化一个安全监视程序，作为安全和不安全操作系统之间的一个安全

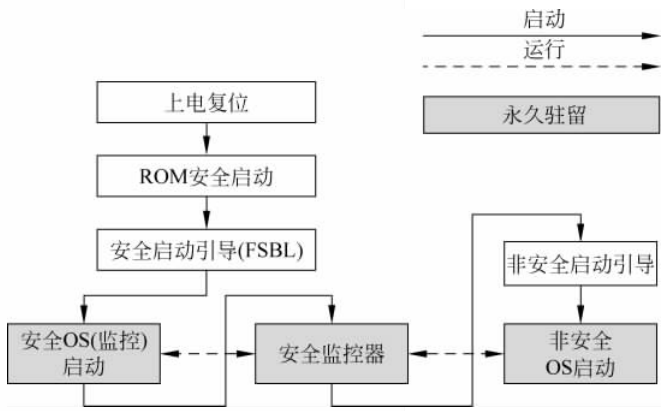

图 3.8　TrustZone 启动序列

“关口”。当安全监视程序启动后，它能产生一个不安全启动引导程序，反过来启动一个不安全的 OS。在初始化不安全 OS 前，安全 OS 定义一组事件，强迫从不安全的 OS 过渡到安全监控程序。可能的事件包括 SMC 指令、IRQ、FIQ 和数据退出。

3.7.1　CPU 安全过渡

安全监控程序调用(Secure Monitor Call，SMC)引起一个安全监控程序异常，它只能在特权模式下使用。在用户模式下，尝试执行这个指令，将引起一个未定义的指令异常。

除了通过一个 SMC 调用进入监控程序模式外，还有一些额外的方法允许用户通过安全监控程序在安全和不安全的世界来回切换，它在这两个域之间作为一个“门卫”。下面是所有可能的进入监控程序模式的方法：

(1) 外部退出句柄；

(2) FIQ 句柄；

(3) IRQ 句柄。

在安全监控程序模式下，处理器总是处于安全状态，而不依赖于 SCR. NS 比特位。

注：在某些情况下，存在 Cortex-A9 TrustZone 冲突的可能性。在安全的世界中，可能触发错误的异步退出。例如，一个安全的管理程序运行代码，其 SCR. NS＝1，并且屏蔽异步退出比特 CPSR. A＝1。这个不安全的代码尝试读和写标记为安全的存储器，结果接收到 AXI DECERR。此时，由于 CPSR. A＝1 屏蔽了异常，所以不发生异常。一旦重新进入安全模式，管理程序切换到其他代码，执行 DSB/ISB，清除 CPSR. A＝0。在这种情况下，只要 CPSR. A＝0，Cortex-A9 处理器仍然记住正在停止的异步外部退出，然后立即产生异常。

此外，不安全的代码可能有多字保存到安全存储器中，引起 L1 缓存的数据修改。当最终替换出这个数据时，在安全模式下，产生异步外部退出。

3.7.2　CP15 寄存器访问控制

在 CP15 下，有一组分组寄存器，意味着相同的寄存器有两个物理拷贝，用于安全和

不安全的模式。当系统不在安全监视程序模式下时，基于 SCR.NS 比特的设置，自动选择物理寄存器。在安全监控程序模式下，总是选择物理寄存器的一个安全版本。表 3.10 给出了 CP15 寄存器的一部分(详细的请参考 ARM 结构参考文档)。

表 3.10 典型的 CP15 寄存器

CP15 寄存器	分组寄存器	允许的访问
c0	CCSELR，缓存大小选择寄存器	只在特权模式读写
c1	SCTLR，系统控制寄存器	只在特权模式读写
	ACTLR，辅助控制器	只在特权模式读写
c2	TTBR0，转换表基地址 0	只在特权模式读写
	TTBR0，转换表基地址 1	只在特权模式读写
	TTBCR，转换表基地址控制	只在特权模式读写
c3	DACR，域访问控制寄存器	只在特权模式读写
c5	DFSR，数据故障状态寄存器	只在特权模式读写
	IFSR，指令故障状态寄存器	只在特权模式读写
	ADFSR，辅助数据故障状态寄存器	只在特权模式读写
	AIFSR，辅助指令故障状态寄存器	只在特权模式读写
c6	DFAR，数据故障地址寄存器	只在特权模式读写
	IFAR，指令故障地址寄存器	只在特权模式读写
c7	PAR，物理地址寄存器(VA 到 PA 翻译)	只在特权模式读写
c10	PRRR，基本的区域重映射寄存器	只在特权模式读写
	NMRR，普通的存储器重映射寄存器	只在特权模式读写
c12	VBAR，向量基地址寄存器	只在特权模式读写

表 3.11 给出了额外的寄存器，在安全和不安全的状态下的访问控制。

表 3.11 CP15 状态控制寄存器

CP15 寄存器	分组寄存器	允许的访问
c1	NSACR，不安全访问控制寄存器	只在特权模式读写，在不安全特权模式下，只读
	SCR，安全配置寄存器	只在特权模式读写
	SDER，安全调试使能寄存器	只在特权模式读写
c12	MVBAR，监视程序向量基地址寄存器	只在特权模式读写

3.7.3 MMU 安全性

Cortex-A9 内的 MMU 带有扩展的 TrustZone 特性，用于提供访问许可检查和额外的地址转换。在每个安全和不安全的世界里，一个单个两级页表保存在主存中，用于控制指令和数据侧的页转换表(Translation Look-Aside Buffer，TLB)的内容。与虚拟地址相关的，由计算所得到的物理地址就存放在 TLB 内，与不安全表标识符(Non-Secure Table Identifier，NSTID)在一起，允许安全和不安全的入口并存。通过 CP15 控制寄存器 c1 的单个比特，使能每个世界的 TLB，它用于为软件提供单个地址转换和保护策略。

当在安全和不安全的世界之间进行过渡时，处理 TLB 和 BTAC 的状态的方法如下：

(1) 当从安全状态执行 BPIALL 时,在不安全的状态下,可能使或者可能不使 BTAC 入口无效;

(2) 当从安全状态执行 TLBIALL 时,不能从安全状态使 BTAC 入口无效;

(3) 从安全状态写 CONTEXTIDR,但是带有 SCR. NS=1,则在不安全状态下使 BTC 入口无效。

由于任何上下文切换将写到 CONTEXTIDR,这样使得不安全的 BTAC 入口无效。而来自安全状态的 BPIALL 不影响不安全 BTAC 入口,因此将来这不是一个问题。

3.7.4 L1 缓存安全性

每个缓存行包含安全和非安全的数据。一个尝试违反安全型访问的结果是引起一个缓存缺失。当缺失时,下一步是到外部存储器。如果 NS 的属性不匹配访问的许可,则返回退出。

3.7.5 安全异常控制

当采纳一个异常时,处理器将被迫转移到一个地址执行,这个地址对应于相应的异常类型。这些地址称为异常向量。默认时,异常向量是 8 个连续字对齐的存储器地址。

对于 Zynq-7000 器件,有三个异常基地址:

(1) 非安全异常基地址,用于非安全状态所有异常的处理;

(2) 安全异常基地址,用于安全状态异常的处理。但是,不在监控程序模式下;

(3) 监控程序异常基地址,用于在监控程序模式下对所有异常进行处理。

3.7.6 CPU 调试 TrustZone 访问控制

四个控制信号控制 CPU 的调试状态:DBGEN、NIDEN、SPIDEN 和 SPNIDEN。这四个控制信号是设备配置接口模块内的一个安全和被保护寄存器的一部分。表 3.12 给出了所支持的最重要的模式。

表 3.12 CPU 调试 TrustZone 访问控制

模　式	DBGEN	NIDEN	SPIDEN	SPNIDEN	描　述
非调试	0	0	0	0	无 CPU 调试
非安全非侵入式调试	1	1	0	0	允许非侵入式调试,比如非安全模式下,跟踪和性能监视程序
非安全侵入式调试	1	1	0	0	允许侵入式调试,比如在非安全模式下,停止处理器
安全非侵入式调试	1	1	0	1	在安全条件下,允许 CPU 跟踪和统计
安全侵入式调试	1	1	1	1	允许用于安全模式下的侵入式调试

3.7.7 SCU 寄存器访问控制

SCU 的非安全访问控制寄存器(SCU Non-Secure Access Control Register,

SNACR),用于控制对 SCU 内每个主元件的全局不安全访问。中断控制器分配器控制寄存器(Interrupt Controller Distributor Control Register,ICDDCR)是一个分组的寄存器,用于控制安全和非安全的访问。

3.7.8 L2 缓存中的 TrustZone 支持

缓存控制器为 L2 缓存和内部缓冲区内所有保存的数据,添加一个 NS 比特位。一个不安全的交易不能访问安全的数据。因此,控制器将安全和不安全的数据看作是两个不同存储器空间的一部分。控制器将对 L2 缓存内安全数据的非安全访问看作一个缺失。对于读传输,缓存控制器给外部存储器发送一个行填充命令,将来自外部存储器的任何安全错误传到处理器。并且,不在 L2 缓存内为其分配缓存行。

下面是关于 L2 内支持 Trustzone 的一些注意事项:

(1) 只能通过带有标记为安全的访问,写 L2 控制寄存器,用来使能或者禁止 L2 缓存;

(2) 只能通过带有标记为安全的访问,写辅助控制寄存器,辅助控制寄存器内的比特[26]用于使能 NS 锁定,该位用于确定,一个非安全的访问是否能修改一个锁定寄存器;

(3) 非安全的维护操作,不能清除或者使安全数据无效。

思考题 3-13:Zynq-7000 的 APU 提供了哪些安全机制,这些安全机制应用在哪些场合?

3.8 应用处理单元复位

3.8.1 复位功能

APU 支持不同的复位模式,可以使用户单独的复位模块的一部分。下面给出了可应用的复位和这些复位的功能。

1. 上电复位

上电复位或者冷复位,应用在开始给系统上电或者通过 PS_POR_B 器件引脚的复位。在这个复位模式中,复位所有 CPU、NEON 协处理器和调试逻辑。

2. 系统复位

系统复位,初始化除调试逻辑外的 Cortex-A9 处理器和 NEON 协处理器。在这个复位过程中,仍然保留断点和监视点。通过 PS_SRST_B 引脚使用这个复位。

3. 软件复位

软件复位或者热复位,用于初始化除调试逻辑外的 Cortex-A9 处理器和 NEON 协处理器。在这个复位过程中,仍然保留断点和监视点。典型地,处理器复位用于复位已经运行了一段时间的系统。通过 A9_CPU_RST_CTRL. A9_RSTx 寄存器,使用这个复位。

4. 系统调试复位

这个复位类似于软件复位。然而,它通过 JTAG 接口触发。

5. 调试复位

这个复位,初始化 Cortex-A9 处理器内的调试逻辑,包括断点和监视点的值。通过 JTAG 接口触发调试复位。

注:(1) Zynq-7000 EPP 器件内的 APU 不支持对 NEON 协处理器独立地复位。

(2) 与上电复位或者系统复位不同,当用户用软件复位单个处理器时,用户必须停止相关的时钟,释放复位,然后重新启动时钟。在一个系统或者上电复位时,硬件自动考虑到这个问题。假设用户想要复位 CPU0,用户必须在 SCLR. A9_CPU_RST_CTRL(其地址位于 0x000244 的位置)寄存器中按顺序设置下面的域:①A9_RST0=1,确认复位 CPU0;②A9_CLKSTOP0=1,停止 CPU0 的时钟;③A9_RST0=0,释放对 CPU0 的复位;④A9_CLKSTOP0=0,重新启动 CPU0 的时钟。

思考题 3-14:Zynq-7000 提供了哪些复位的机制,用于对 APU 进行复位操作?

3.8.2 复位后的 APU 状态

表 3.13 总结了复位后的 APU 状态。

表 3.13 复位后的 APU 状态

功 能	复位后的状态
CPU1	保持在 WFE 状态,同时执行位于地址 0xFFFFFE00 到 0xFFFFFFF0 的代码
L1 缓存	禁止
有效	未知(要求使用前无效)
MMU	禁止
SCU	—
地址过滤	4G 地址空间的高和低 1M 地址映射到 OCM,剩下的地址连接到 L2
L2 缓存	禁止
L2 等待状态	标记 RAM 和数据 RAM 等待状态都是 7-7-7,用于建立延迟、写延迟和读访问延迟
有效	未知(要求使用前无效)

3.9 功耗考虑

APU 结合很多特性用于改善它的动态功耗的效率:

(1) 其中的一个 CPU 能进入休眠模式,并且可以从事件和中断中唤醒;

(2) 当 CPU 处于待机模式时,L2 缓存可以进入待机模式;

(3) 模块内的所有子模块广泛地使用门控技术;

(4) 准确的分支和返回预测,减少了取指不正确指令和译码操作的个数;

(5) 物理寻址的缓存减少了缓存刷新和重新填充的次数,降低了系统内的功耗;

(6) CPU 实现 micro TLB,用于转换本地地址,降低了转换和保护查找所消耗的

功耗；

(7) 按顺序访问标记 RAM 和数据 RAM，这样避免访问不想要的数据 RAM，因此减少了不必要的功耗；

(8) 通过利用存储器顺序访问的本质，减少全缓存读的次数，因此降低了 L1 缓存的功耗，如果一个缓存读是前面的缓存读的继续，并且地址是在相同的缓存行内，只访问先前读数据的 RAM 组；

(9) 在 4 个 BTAC 入口内，如果一个指令循环相适配，则关闭指令缓存访问，用于降低功耗；

(10) 由 CPU 动态地控制 NEON 引擎的时钟，当发送 NEON 指令时，NEON 引擎才会得到时钟。

注：当 PS 上电时，不能关闭到 APU 或者任何它子模块的电源。

3.9.1 待机模式

在待机操作模式下，器件仍然上电，但是关闭它的大部分时钟。这意味着处理器处于静态时，只有漏电流和少量用于寻找唤醒条件逻辑的时钟消耗功率。

通过使用 WFI 或者 WFE 指令，CPU 进入这个模式。推荐在执行 WFI 或者 WFE 指令前，使用 DSB 存储器屏障，保证完全停止存储器交易。

处理器停止执行应用程序，直到检测到唤醒事件。唤醒条件取决于入口条件。对于 WFI 指令，一个中断或者外部调试请求唤醒处理器。对于 WFE 指令，存在大量指定的事件，包括多核系统内另一个处理器执行 SEV 指令。来自 SCU 的一个请求，也可以唤醒多核系统内用于缓存一致性操作的时钟。这意味着，待机状态下的一个处理器缓存一直和其他处理器的缓存保持一致。一个处理器复位总是强迫处理器从待机条件退出。

通过设置 L2 控制器电源控制寄存器的 0 比特，使能 L2 缓存控制器的待机模式。这个模式和处理器的等待状态一起使用，用于驱动控制器。在进入等待状态前，Cortex-A9 处理器必须在 SCU 的 CPU 供电状态寄存器内设置它的状态位，发送信号进入休眠模式。然后，Cortex-A9 处理器执行 WFI 或者 WFE 入口指令。通过正在退出低功耗模式的 Cortex-A9 处理器，读取 SCU CPU 电源状态寄存器比特位。该比特位用于在执行处理器复位设置前，确定处理器的状态。

如果多核处理器系统处于待机模式下，SCU 则发信号到 L2 缓存控制器，用来门控时钟。当 L2 变成空闲状态时，控制器用来实现门控。任何从 SCU 到 L2 的交易，将重新启动时钟。并且，在 2～3 个时钟周期延迟后触发一个响应。

3.9.2 在 L2 控制器内的动态时钟门控

L2 控制器的电源控制寄存器的[1]比特，用于使能控制器内的动态时钟门控特性。如果使能这个特性，在空闲状态时，缓存控制器将它的时钟停止 32 个时钟周期。控制器停止时钟，直到从接口上有来自 SCU 的交易为止。如果在这个接口上检测到一个交易，它重新启动它的时钟。并且，在 2～3 个时钟周期延迟后，接受新的交易。

3.10 系统地址分配

Zynq-7000为系统内的各个模块分配了地址空间，这对于软件编程是非常重要的。

3.10.1 地址映射

表3.14给出了系统级的地址映射。阴影部分表示保留的地址范围，不能访问该地址范围。表3.15标识出保留的地址范围。

表3.14 系统级地址映射

地址范围	CPU和ACP	AXI_HP	其他总线主设备(1)	注意
0000_0000～0003_FFFF(2)	OCM	OCM	OCM	地址没有被SCU过滤，OCM被映射到低地址范围
	DDR	OCM	OCM	地址被SCU过滤，OCM被映射到低地址范围
	DDR			地址被SCU过滤，OCM没有被映射到低地址范围
				地址没有被SCU过滤，OCM没有被映射到低地址范围
0004_0000～000F_FFFF(3)	DDR	DDR	DDR	地址被SCU过滤
		DDR	DDR	地址没有被SCU过滤
0010_0000～3FFF_FFFF	DDR	DDR	DDR	可访问所有互联主设备
4000_0000～7FFF_FFFF	PL		PL	到PL的通用端口#0，M_AXI_GP0
8000_0000～BFFF_FFFF	PL		PL	到PL的通用端口#1，M_AXI_GP1
E000_0000～E02F_FFFF	IOP		IOP	I/O外设寄存器，见表3.16
E100_0000～E5FF_FFFF	SMC		SMC	SMC存储器，见表3.17
F800_0000～F800_0BFF	SLCR		SLCR	SLCR寄存器，见表3.18
F800_1000～F880_FFFF	PS		PS	PS系统寄存器，见表3.19
F890_0000～F8F0_2FFF	CPU			CPU私有寄存器，见表3.20
FC00_0000～FDFF_FFFF(4)	四-SPI		四-SPI	用于线性模式的四-SPI线性地址
FFFC_0000～FFFF_FFFF(2)	OCM	OCM	OCM	OCM被映射到高地址范围
				OCM没有被映射到高地址范围

(1) 其他总线主设备包括S_AXI_GP接口，器件配置接口(DevC)、DAP控制器、DMA控制器和带有本地DMA单元的各种控制器(以太网、USB和SDIO)。

(2) OCM被分成4个64KB区域。每个区域独立映射到低或者高地址范围。但是，在同一时间并不是所有。此外，SCU能过滤到DDR DRAM控制器的OCM低地址范围。

(3) 在一个主设备访问DDR存储器空间前，必须使能DDR存储器控制器。如果禁止控制器或者没有产生功能，则在DDR存储器空间内的读或者写将挂起互联，并且引起看门狗定时器复位。除了CPU和ACP外，任何主设备不可以使用DDR的低512KB空

间。当使能对 SCU 低 1MB(地址过滤的颗粒度)的地址过滤时,CPU 和 ACP 只能访问低 512KB 空间。

(4) 当使用单个设备时,它必须连接到 QSPI 0。在这种情况下,地址映射起始于 FC00_0000,最大到 FCFF_FFFF(16MB)。当使用两个设备时,这两个设备必须有相同的容量。对于两个设备的地址映射关系,取决于设备的大小和它们的连接配置。对于共享 4 比特位的并行 I/O 总线,QSPI 0 器件的地址起始于 FC00_0000,最大到 FCFF_FFFF(16MB)。QSPI 1 设备的地址起始于 FD00_0000,最大到 FDFF_FFFF(16MB)。如果第一个设备小于 16MB,则在两个设备间存在一个存储器的空隙。对于 8 比特的双堆栈模式(8 位总线)。存储器地址映射空间,连续的从 FC00_0000 到最大的 FDFF_FFFF(32MB)。

表 3.15 系统级地址映射(保留地址)

地 址 范 围	CPU 和 ACP	AXI_HP	其他总线主设备	注意
C000_0000～DFFF_FFFF				保留
E030_0000～E0FF_FFFF				保留
E600_0000～F7FF_FFFF				保留
F800_0C00～F800_0FFF				保留
F801_0000～F88F_FFFF				保留
F8F0_3000～FBFF_FFFF				保留
FE00_0000～FFFB_FFFF				保留

PL AXI 接口注意事项:在 Zynq-7000 内,有两个互联接口 M_AXI_GP{1,0}引入到 PL。PS 内的主设备可以对每个端口寻址。如表 3.14 所示,每个端口分配 1GB 的系统地址空间。M_AXI_GP 的地址直接来自 PS,M_AXI_GP 不再以自己的方式重新映射到 PL。这些范围外的地址不会出现在 PL。

就地执行功能的设备:下面的设备具有就地执行功能:DDR、OCM、SMC SRAM/NOR、四-SPI(线性地址模式)、M_AXI_GP{1,0}(带有合适 PL 控制器的 PL BRAM)。

3.10.2 系统总线主设备

除了 CPU 有私有总线用于访问它们的私有定时器、中断控制器和共享的 L2 缓存/SCU 寄存器外,CPU 和 AXI_ACP 可以看到相同的存储器映射。AXI_HP 接口为 DDR DRAM 和 OCM 提供高带宽。其他的系统总线主设备包括:

(1) DMA 控制器;

(2) 设备配置接口(DevC);

(3) 调试访问端口(DAP);

(4) 连接到 AXI 通用端口上的 PL 总线主设备控制器;

(5) 带有本地 DMA 的 AHB 总线主设备端口(以太网、USB 和 SDIO)。

3.10.3 I/O 外设

如表 3.16 所示,通过一个 32 位的 APB 总线访问 I/O 外设寄存器。

表 3.16　I/O 外设寄存器映射

寄存器基地址	描　　述
E000_0000 和 E000_1000	UART 控制器 0 和 UART 控制器 1
E000_2000 和 E000_3000	USB 控制器 0 和 USB 控制器 01
E000_40000 和 E000_5000	I^2C 控制器 0 和 I^2C 控制器 1
E000_6000 和 E000_7000	SPI 控制器 0 和 SPI 控制器 1
E000_8000 和 E000_9000	CAN 控制器 0 和 CAN 控制器 1
E000_A000	GPIO 控制器
E000_B000 和 E000_C000	以太网控制器 0 和以太网控制器 1
E000_D000	四-SPI 控制器
E000_E000	静态存储器控制器(SMC)
E010_0000 和 E010_1000	SDIO 控制器 0 和 SDIO 控制器 1

3.10.4　SMC 存储器

通过一个 32 位的 AHB 总线访问 SMC 存储器。表 3.17 列出了 SMC 控制器寄存器。

表 3.17　SMC 存储器地址映射

寄存器基地址	描　　述
E100_0000	SMC NAND 存储器地址范围
E200_0000	SMC SRAM/NOR CS0 存储器地址范围
E400_0000	SMC SRAM/NOR CS1 存储器地址范围

3.10.5　SLCR 寄存器

系统级控制寄存器(System-Level Control Register,SLCR)由各种寄存器组成,用于控制 PS 的行为。这些寄存器使用加载和保存指令,通过中央互联进行访问。表 3.18 给出了 SLCR 寄存器的映射。

表 3.18　SLCR 寄存器映射

寄存器基地址	描　　述
F800_0000	SLCR 写保护锁定和安全
F800_0100	时钟控制和状态
F800_0200	复位控制和状态
F800_0300	APU 控制
F800_0400	TrustZone 控制
F800_0500	CoreSight SoC 调试控制
F800_0600	DDR DRAM 控制器
F800_0700	MIO 引脚配置
F800_0800	MIO 并行访问
F800_0900	杂项控制
F800_0A00	片上存储器(OCM)控制
F800_0B00	用于 MIO 引脚和 DDR 引脚的 I/O 缓冲区

3.10.6 杂项 PS 寄存器

如表 3.19 所示，通过 32 位的 AHB 总线访问 PS 系统寄存器。

表 3.19 SLCR 寄存器映射

寄存器基地址	描　　述
F800_1000，F800_2000	三重定时器计数器 0 和三重定时器计数器 1
F800_3000	当安全时，DMAC
F800_4000	当非安全时，DMAC
F800_5000	系统看门狗定时器(SWDT)
F800_6000	DDR DRAM 控制器
F800_7000	设备配置接口(DevC)
F800_8000	AXI_HP 0 高性能 AXI 接口 w/FIFO
F800_9000	AXI_HP 1 高性能 AXI 接口 w/FIFO
F800_A000	AXI_HP 2 高性能 AXI 接口 w/FIFO
F800_B000	AXI_HP 3 高性能 AXI 接口 w/FIFO
F800_C000	片上存储器 OCM
F800_D000	保留
F880_0000	CoreSight 调试控制

3.10.7 CPU 私有总线寄存器

表 3.20 给出的寄存器只能是 CPU 通过 CPU 私有总线进行访问。ACP 不能访问任何 CPU 的私有寄存器。私有 CPU 寄存器用于在 APU 中控制子系统。

表 3.20 CPU 私有寄存器映射

寄存器基地址	描　　述
F890_0000～F89F_FFFF	顶层互联配置和全局编程器查看(GPV)
F8F0_0000～F8F0_00FC	SCU 控制和状态
F8F0_0100～F8F0_01FF	中断控制器 CPU
F8F0_0200～F8F0_02FF	全局定时器
F8F0_0600～F8F0_06FF	私有定时器和私有看门狗定时器
F8F0_1000～F8F0_1FFF	中断控制器分配器
F8F0_2000～F8F0_2FFF	L2 缓存控制器

思考题 3-15：说明 Zynq-7000 内系统地址的空间分配。

3.11 中断

图 3.9 给出了中断控制器的系统级中断环境和功能。PS 基于 ARM 结构，使用两个 Cortex-A9 处理器和 GIC pl390 中断控制器。

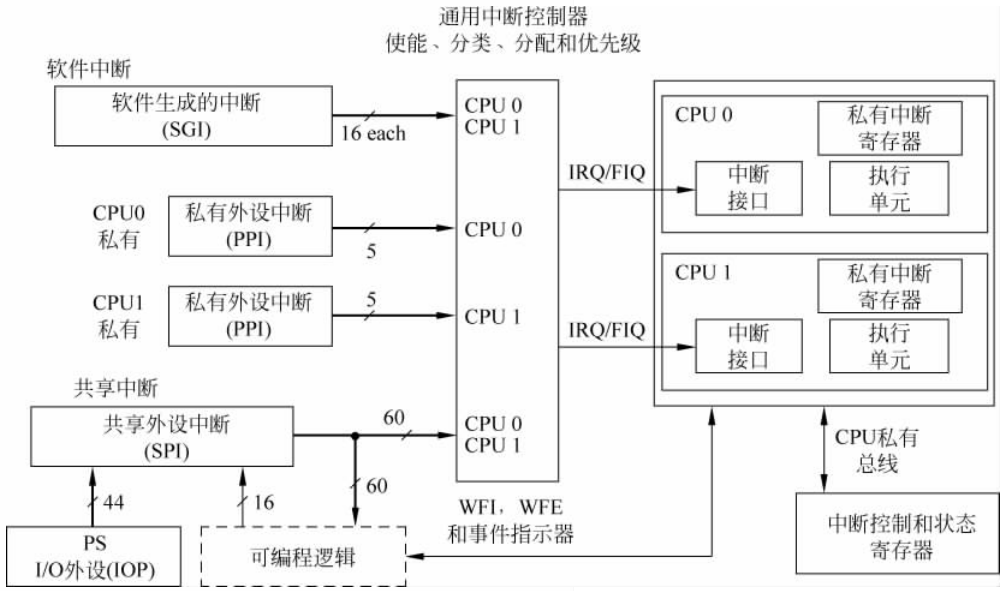

图 3.9　系统级中断级结构图

3.11.1　中断环境

中断结构和 CPU 紧密相关，接受来自 I/O 外设(IOP)和可编程逻辑(PL)的中断。

1. 私有、共享和软件中断

每个 CPU 都有一组私有外设中断(Private Peripheral Interrupt，PPI)，它通过使用分组的寄存器进行私有访问。PPI 包括全局定时器、私有看门狗定时器、私有定时器和来自 PL 的 FIQ/IRQ。软件产生的中断(Software Generated Interrupt，SGI)连接到其中一个或者所有的 CPU。通过写 ICDSGIR 寄存器，产生软件中断。通过 PS 和 PL 内各种 I/O 和存储器控制器，产生共享的外设中断(Shared Peripheral Interrupt，SPI)。图 3.10 给出了中断源，这些中断源连接到其中一个 CPU 或者所有的 CPU。来自 PS 外设的共享外设中断，也连接到 PL。

2. 通用中断控制器

通用中断控制器(Generic Interrupt Controller，GIC)是核心资源，用于管理来自 PS 或者 PL 的中断，这些中断发送到 CPU。按照编程的行为，当 CPU 接口接受下一个中断时，控制器使能、禁止、屏蔽和优先级设置中断源。并且，将它们发送到所选择的某个或者所有的 CPU。此外，控制器支持安全扩展，用于实现一个安全意识系统。

控制器是非向量的，其结构基于 ARM 通用中断控制器结构 V1.0(GIC v1)。

为了避免互联内暂时的阻塞或者其他瓶颈，通过 CPU 私有总线访问寄存器，以实现快速地读/写响应。

将带有最高优先级的一个中断分配到单个的 CPU 时，中断分配器将所有的中断源集中在一起。在一个时刻，硬件确保针对几个 CPU 的一个中断，只能被一个 CPU 采纳。

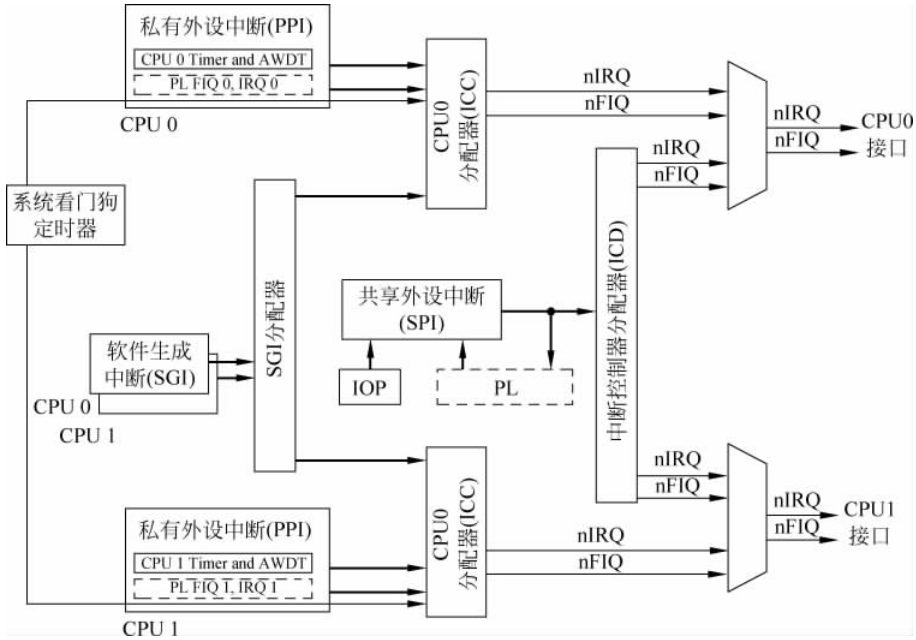

图 3.10　中断控制器的块图

通过一个唯一的中断 ID 号识别所有的中断源。所有的中断源都有它们自己可配置的优先级和目标 CPU 的列表。

3. 复位和时钟

通过写 SLCR 内的 A9_CPU_RST_CTRL 寄存器的 PERI_RST 比特位，复位子系统。这样，就可以复位中断控制器。同样地，复位信号也复位 CPU 私有定时器和私有看门狗定时器(AWDT)。

中断控制器工作在 CPU_3x2x 时钟域(一半的 CPU 频率)。

3.11.2　中断控制器的功能

1. 软件中断

如表 3.21 所示，存在 16 个软件产生的中断。通过向 ICDSGIR 寄存器写入 SGI 中断号，以及指定目标 CPU，来产生一个软件中断。通过 CPU 私有总线，实现这个写操作。CPU 能中断自己，或者其他 CPU，或者所有的 CPU。通过读 ICCIAR 寄存器或者向 ICDICPR(Interrupt Clear-Pending)寄存器相应的比特位写 1，可以清除中断。

所有的 SGI 为边沿触发。用于 SGI 的敏感性是固定的，不能修改。ICDICFR0 寄存器是只读寄存器。

2. CPU 私有外设中断(PPI)

每个 CPU 连接一个私有的 5 个共享的外设中断。表 3.22 给出了 CPU 的私有外设中断。

表 3.21 软件中断(Software Generated Interrupt,SGI)

名 字	SGI#	ID#	类 型
软件 0	0	0	上升沿
软件 1	1	1	上升沿
…	～	～	…
软件 15	15	15	上升沿

表 3.22 私有外设中断

名 字	PPI#	ID#	类 型	描 述
保留	～	26:16	～	保留
全局定时器	0	27	上升沿	全局定时器
nFIQ	1	28	活动低电平(在 PS-PL 接口,活动高)	来自 PL 的快速中断
CPU 私有定时器	2	29	上升沿	来自 CPU 定时器的中断
AWDT{0,1}	3	30	上升沿	用于每个 CPU 的私有看门狗定时器
nIRQ	4	31	活动低电平(在 PS-PL 接口,活动高)	来自 PL 的中断信号

所有中断的敏感类型是固定的,不能改变。

注:将来自 PL 的快速中断信号 FIQ 和中断信号 IRQ 翻转,然后送到中断控制器中。因此,尽管在 ICDICFR1 寄存器内反映它们是活动低敏感信号,但是在 PS-PL 接口,为高电平活动。

3. 共享外设中断(SPI)

来自不同模块一组大约 60 个中断能连接到一个/两个 CPU 或者 PL。中断控制器用于管理中断的优先级和接受用于 CPU 的这些中断。

默认地,所有共享外设中断类型的复位是一个活动高电平。然而,软件使用 ICDICFR2 和 ICDICFR5 寄存器编程中断 32、33 和 92 为上升沿敏感。表 3.23 列出了共享外设中断。

表 3.23 PS 和 PL 共享外设中断(SPI)

源	中断名字	中 断		类型	PS-PL 信号名字	I/O
		SPI ID#	PPI			
APU	L1 缓存	33,32	～	上升沿	～	～
	L2 缓存	34	～	高电平	～	～
	OCM	35	～	高电平	～	～
保留	～	36	～	～	～	～
PMU	PMU[1,0]	37,38	～	高电平	～	～
XADC	XADC	39	～	高电平	～	～
DVI	DVI	40	～	高电平	～	～
SWDT	SWDT	41	～	高电平	～	～
定时器	TTC0	43: 42	～	高电平	～	～
保留	～	44	～	～	～	～

续表

源	中断名字	中断		类型	PS-PL 信号名字	I/O
		SPI ID#	PPI			
DMAC	DMAC 退出	45	～	高电平	IRQP2F[28]	输出
	DMAC[3:0]	49：46	～	高电平	IRQP2F[23:20]	输出
存储器	SMC	50	～	高电平	IRQP2F[19]	输出
	四-SPI	51	～	高电平	IRQP2F[18]	输出
调试	CTI	～	～	高电平	IRQP2F[17]	输出
IOP	GPIO	52	～	高电平	IRQP2F[16]	输出
	USB 0	53	～	高电平	IRQP2F[15]	输出
	以太网 0	54	～	高电平	IRQP2F[14]	输出
	以太网 0 唤醒	55	～	高电平	IRQP2F[13]	输出
	SDIO 0	56	～	高电平	IRQP2F[12]	输出
	I^2C 0	57	～	高电平	IRQP2F[11]	输出
	SPI 0	58	～	高电平	IRQP2F[10]	输出
	UART 0	59	～	高电平	IRQP2F[9]	输出
	CAN 0	60	～	高电平	IRQP2F[8]	输出
PL	FPGA[7:0]	68:61	～	高电平	IRQP2F[7:0]	输入
定时器	TTC 1	71:69	～	高电平	～	～
DAMC	DMAC[7:4]	75:72	～	高电平	IRQP2F[27:24]	输出
IOP	USB 1	76	～	高电平	IRQP2F[7]	输出
	以太网 1	77	～	高电平	IRQP2F[6]	输出
	以太网 1 唤醒	78	～	高电平	IRQP2F[5]	输出
	SDIO 1	79	～	高电平	IRQP2F[4]	输出
	I^2C 1	80	～	高电平	IRQP2F[3]	输出
	SPI 1	81	～	高电平	IRQP2F[2]	输出
	UART 1	82	～	高电平	IRQP2F[1]	输出
	CAN 1	83	～	高电平	IRQP2F[0]	输出
PL	FPGA[15:8]	91:84	～	高电平	IRQP2F[15:8]	输入
SCU	奇偶校验	92	～	上升沿	～	～
保留	～	95:93	～	～	～	～
PL	FIQ 到 CPU1	～	CPU 1 nFIQ		IRQP2F[19]	输入
	FIQ 到 CPU0	～	CPU 0 nFIQ		IRQP2F[18]	输入
	IRQ 到 CPU1	～	CPU 1 nIRQ		IRQP2F[17]	输入
	IRQ 到 CPU0	～	CPU 0 nIRQ		IRQP2F[16]	输入

4. 寄存器

中断控制器 CPU(Interrupt Controller CPU,ICC)和中断控制器分配器(Interrupt Controller Distributer,ICD)寄存器是 pl390 GIC 寄存器集。这里有 60 个共享外设中断。这远比 pl390 能支持的中断要少得多。所以,在 ICD 内,比 pl390 内的中断使能、

状态和优先级和处理器目标寄存器要少的很多。表 3.24 给出了 ICC 和 ICD 寄存器的列表。

表 3.24　中断控制寄存器

名　　字	寄存器描述	写保护锁定
中断控制器 CPU(ICC)		
ICCICR	CPU 接口控制	是,除了 EnableNS
ICCPMR	中断优先级屏蔽	～
ICCBPR	用于中断优先级的二进制小数点	～
ICCIAR	中断响应	～
ICCEOIR	中断结束	～
ICCRPR	运行优先级	～
ICCHPIR	最高待处理的中断	～
ICCABPR	别名非安全的二进制小数点	～
中断控制器分配器(ICD)		
ICCDDCR	安全/非安全模式选择	是
ICDICTR,ICDIIDR	控制器实现	～
ICDISR[2:0]	中断安全	是
ICDISER[2:0] ICDICER[2:0]	中断设置使能和清除使能	是
ICDISPR[2:0] ICDICPR[2:0]	中断设置待处理和清除待处理	是
ICDABR[2:0]	中断活动	～
ICDIPR[23:0]	中断优先级,8 比特位域	是
ICDIPTR[23:0]	中断处理器目标,8 比特位域	是
ICDICFR[5:0]	中断敏感类型,2 比特域(电平/边沿,正/负)	是
PPI 和 SPI 状态		
PPI_STATUS	PPI 状态	～
SPI_STATUS[2:0]	SPI 状态	～
软件中断(SGI)		
ICDSGIR	软件产生的中断	～
禁止写访问(SLCR 寄存器)		
APU_CTRL	CFGSDISABLE 比特位禁止一些写访问	～

中断控制器提供了工具,用于阻止对关键配置寄存器的写访问。通过给 APU_CTRL[CFGSDISABLE]比特位写 1,实现这个功能。APU_CTRL 寄存器是 Zynq 系统级控制寄存器组的一部分。这个控制寄存器,用于对中断控制寄存器的安全写行为。

如果用户想要设置 CFGSDISABLE 比特位,推荐在用户软件启动过程中进行操作。用户软件启动过程发生在软件配置中断控制寄存器之后。只能在上电复位时,清除

CFGSDISABLE 比特位。当设置完成 CFGSDISABLE 后，它将保护寄存器的比特位改为只读。因此，不能改变这些安全中断的行为。甚至在安全域内出现“流氓”代码执行的时候也不能改变。

3.11.3 编程模型

1. 中断优先级

给所有的中断请求(PPI、SGI 和 SPI)分配独一无二的 ID 号。控制器使用 ID 号进行仲裁。中断分配器保留了用于每个 CPU 的待处理中断。将这些中断发送给 CPU 前，选择最高优先级的中断。通过选择最低的 ID，来解决相同优先级的中断。

将优先级逻辑物理复制，这样可以为每个 CPU 能同时选择最高优先级的中断。中断分配器保留了中断、处理器和活动信息的集中列表。并且，负责触发到 CPU 的软件中断。

将 SGI 和 PPI 分配器寄存器分组，为每个连接的处理器提供单独的复制。在一个时刻，硬件保证面向几个 CPU 的一个中断，只能被一个 CPU 所采纳。

中断分配器将待处理的最高优先级中断发送到 CPU。它接收中断响应后返回的信息，来改变相应中断的状态。只有响应中断的 CPU 才能结束中断。

2. 中断处理

在 ARM 文档 IHI0048B_gic_ architecture_specification. pdf 中描述了，当一个 IRQ 线无效时，GIC 对一个待处理中断的响应过程。

如果在 GIC 内的中断正在等待处理时，不继续确认 IRQ 线，则 GIC 内的中断变成不活动的(即 CPU 从来看不到这个中断)。

如果在 GIC 内的中断是活动的(因为 CPU 接口已经响应了中断)，则软件中断服务程序首先检查 GIC 寄存器来确定原因。然后，轮询 I/O 外设中断状态寄存器。

3. ARM 编程主题

ARM GIC 结构规范包含下面这些编程主题：

(1) GIC 寄存器访问；

(2) 分配器和 CPU 接口；

(3) GIC 安全性扩展影响；

(4) 保护和恢复控制器状态。

4. 传统中断和安全性扩展

当使用传统的中断(IRQ，FIQ)时，在安全模式下(通过 ICCICR[AckCtl]=1)，一个中断控制器访问 IRQ 和 FIQ。当读中断 ID 时，偶尔会产生竞争条件。在 IRQ 句柄内看 FIQ ID，也存在一个冒险。这是由于 GIC 只知道句柄正在从什么安全状态读取，而不知道句柄的类型。

这里有两个可用的解决方案：

(1) 只有信号 IRQ 到一个可重入的 IRQ 句柄,并使用 GIC 内的抢占特性;

(2) 在 ICCICR[AckCtl]=0 情况下,使用 FIQ 和 IRQ。并且,在非安全模式下,使用 TLB 表管理 IRQ,以及在安全模式下管理 FIQ。

思考题 3-16: Zynq-7000 内的 APU 提供了哪些中断能力? 每种类型的中断实现什么功能?

3.12 定时器

每个 Cortex-A9 处理器有它自己的私有 32 位定时器和 32 位看门狗定时器。所有处理器共享一个全局 64 位定时器。这些定时器总是工作在 1/2 的 CPU 频率(CPU_3x2x)。

在系统级上,有一个 24 位的看门狗定时器和两个 16 位 3 重定时器/计数器(Triple Timer/Counter,TTC)。系统级的看门狗定时器工作在 1/4 或者 1/6 的 CPU 工作频率(CPU_1x),或从 MIO 引脚或 PL 的时钟驱动。两个三重定时器/计数器总是驱动在 1/4 或者 1/6 的 CPU 工作频率(CPU_1x),用来计算来自 MIO 引脚或者 PL 的信号脉冲宽度。

图 3.11 给出了系统定时器之间的关系结构。

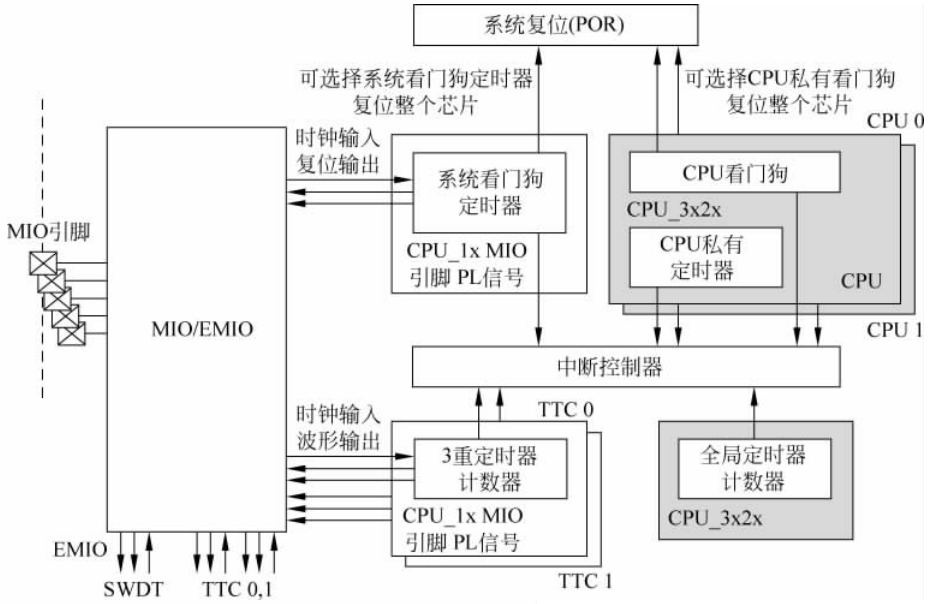

图 3.11 定时器系统结构

3.12.1 CPU 私有定时器和看门狗定时器

定时器和看门狗模块有下面的特性:

(1) 32 位计数器,当达到零时产生一个中断;

(2) 8 位预分频器,能够更好地控制中断周期;

(3) 可配置的一次性或者自动加载模式;

(4) 配置用于计数器的启动值；

(5) SCU 时间间隔可通过下式进行计算：

时间间隔 = [(预分频器的值 + 1) × (加载值 + 1)]/CPU 时钟周期

所有私有定时器和看门狗定时器总是工作在 1/2 的 CPU 频率(CPU_3x2x)。

表 3.25 给出了 CPU 私有和看门狗定时器的寄存器概述。

表 3.25 CPU 私有定时器计数器概述

功 能	名 字	概 述
CPU 私有定时器		
重加载和当前值	定时器加载 定时器计数器	递减器重新加载值 递减当前值
控制和中断	定时器控制 定时器中断	使能、自动重加载、IRQ、预分频器、中断状态
CPU 私有看门狗		
重加载和当前值	看门狗加载 看门狗计数器	递减器重新加载值 递减当前值
控制和中断	看门狗控制 看门狗中断	使能、自动重加载、IRQ、预分频器、中断状态(注意：不能禁止看门狗)
复位状态	看门狗复位状态	复位状态是看门狗到达 0 时的结果 只有上电复位才能清除，这样就能告诉软件，复位是否由看门狗引起
禁止	看门狗禁止	通过写两个指定的字序列，禁止看门狗

3.12.2 全局定时器

全局定时器(Global Timer Counter，GTC)是一个 64 位的递增定时器，带有自动递增的特性。全局定时器是存储器映射的，与私有定时器有相同的地址空间。只有在安全状态下复位时，才可以访问全局定时器。所有 Cortex-A9 处理器均可以访问全局定时器。每个 Cortex-A9 处理器有一个 64 位比较器。当全局定时器到达比较器的值时，用于确认一个私有中断。

全局定时器，总是工作在 1/2 的 CPU 时钟频率(CPU_3x2x)。表 3.26 给出了全局定时器的寄存器列表。

表 3.26 全局定时器寄存器概述

功 能	名 字	概 述
全局定时器(GTC)		
当前值	全局定时器计数器	递增当前的值
控制和中断	全局定时器控制 全局中断	使能定时器、使能比较器、IRQ、自动递增、中断状态
比较器	比较器值 比较器递增	比较器当前的值 用于比较器的递增值
	全局定时器禁止	通过写两个指定的字序列，禁止看门狗

3.12.3 系统看门狗定时器

除了两个 CPU 私有定时器外，还有一个系统看门狗定时器（System Watchdog Timer，SWDT），用于发信号指示灾难性的系统失败。例如，PS PLL 失败。不像 AWDT，SWDT 可以从一个外部设备或者 PL 运行一个时钟。并且，为一个外部设备或者 PL 提供一个复位输出。其特点主要包括：

（1）一个内部的 24 位计数器。

（2）可选择的时钟输入：①内部的 PS 总线时钟（CPU_1x）；②内部时钟（来自 PL）；③外部时钟（来自 MIO）。

（3）在超时时，输出一个或者组合：①系统中断（PS）；②系统复位（PS，PL，MIO）。

（4）可编程的超时周期：超时范围为 32、760～268、431、360 时钟周期（在 100MHz 下，330μs～2.7s）。

（5）在超时时，可编程输出信号周期：①系统中断脉冲 4～32 个时钟周期（在 100MHz 下，40～320ns）；②系统复位脉冲 2～256 个时钟周期（在 100MHz 下，20ns～2.6μs）。

图 3.12 给出了 SWDT 的结构。

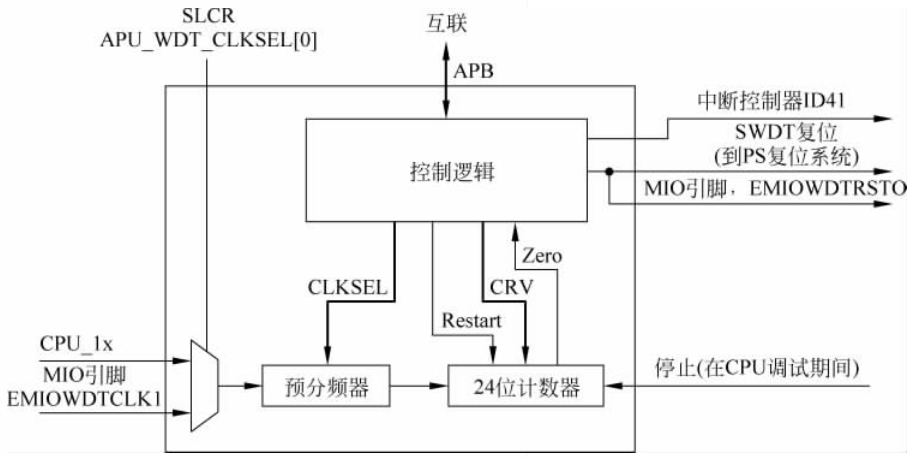

图 3.12 系统看门狗定时器块图

（1）SLCR 可编程寄存器（MIO 控制 APU_WDT_CLKSEL）选择时钟输入。

（2）SWDT 可编程寄存器设置 CLKSEL 和 CRV。

（3）信号 restart 引起 24 位计数器重新加载 CRV 的值和重新启动计数。

（4）在 CPU 调试期间，信号 halt 引起计数器停止（和 AWDT 相同行为）。

控制逻辑块通过 APB 接口连接到系统互联。每个来自 APB 的写数据都有一个关键字域，它必须匹配寄存器的关键字。只有匹配时，才能写寄存器。

当它内部的 24 位计数器达到零时，零模式寄存器用于控制 SWDT 的行为。如果设置 WDEN 和 IRQEN，当收到 zero 信号时，在 IRQLN 周期后，控制逻辑块确认中断输出信号。如果设置了 WDEN 和 RSTLN，在 RSTLN 时钟周期后，控制逻辑块也确认复位输出信号。

通过在 swdt. CONTROL[CLKSET]和 swdt. CONTROL[CRV]内设置重加载的

值，计数器控制寄存器设置超时周期，用于控制预分频器和24位计数器。

重新启动寄存器，用于重新启动计数过程。用匹配的关键字写这个寄存器，将使得预分频器和24位的计数器将在CRV信号重新加载值。

状态寄存器指示24位计数器是否达到零。不考虑零模式寄存器内的WDEN比特位，如果没到零，并且出现了所选择的时钟源，则24位计数器总是向下计数到零。一旦计数到达零，则设置状态寄存器的WDZ比特。并且，一直保持该设置，直到24位计数器重新启动为止。

预分频器模块将对选择的时钟输入频率分频。在每个上升时钟沿，采样CLKSEL信号。

内部的24位计数器计数到零，然后保持在零，一直到计数器重新启动为止。当计数器到达零时，zero输出信号为高。

表3.27给出了SWDT的寄存器描述。

表3.27 系统看门狗定时器寄存器概述

功 能	名 字	描 述
零模式	swdt.MODE	使能SWDT，使能在超时时，中断和复位输出，以及设置输出脉冲长度
重加载值	swdt.CONTROL	在超时时，为预分频器和24位计数器设置重加载值
重新启动	swdt.RESTART	重新加载和重新启动预分频器和24位计数器
状态	swdt.STATUS	指示看门狗达到零

下面给出了使能系统看门狗定时器的控制序列。

(1) 选择时钟输入源(SLCR[WDT_CLK_SEL[SEL]]比特位)：在处理这一步前，确认禁止SWDT(SWDT[MODE[WDEN]=0)。当SWDT使能后，改变时钟输入源，将导致不可预测的结果。

(2) 设置超时周期(计数器控制寄存器)：必须设置SWDT[CONTROL[CKEY]]域为0x248。这样是为了对这个寄存器进行写操作。

(3) 使能计数器，使能输出脉冲，设置输出脉冲长度(零模式寄存器)：必须设置SWDT[MODE[ZKEY]]域为0xABC。这样是为了对这个寄存器进行写操作，确保IRQLN和RSTLN满足指定的最小值。

(4) 为了用不同的设置运行SWDT，首先禁止定时器(SWDT[MODE[WDEN]]比特)，然后重复以上的步骤。

3.12.4 三重定时器/计数器

1. 三重定时器/计数器概述

三重定时器计数器(Triple Timer Counter，TTC)包括三个独立的定时器/计数器。由于是用单个的APB接口访问它们，因此三个定时器/计数器必须有相同的安全状态。在PS中，有两个TTC模块，总共6个定时器/计数器。保留一个TTC，用于安全软件；另一个由安全软件或者非安全软件共享。当不使用TrustZone时，所有的TTC模块均可以用于非安全软件。

三重定时器/计数器的特性包括：

(1) 三个单独的 16 位预分频器和 16 位的向上/向下计数器。

(2) 可选择的时钟：①内部的 PS 总线时钟(CPU_1x)；②内部时钟(来自 PL)；③外部时钟(来自 MIO)。

(3) 三个中断，每个用于一个计数器。

(4) 溢出、在规定的间隔或者计数器匹配的可编程的值。

(5) 通过 MIO 产生到 PL 的波形输出(如 PWM)。

图 3.13 给出了 TTC 的块图。slcr.PIN_MUX 寄存器控制定时器/时钟 0 的时钟和波形输出的复用。如果没有进行选择，则默认变成 EMIO 接口。

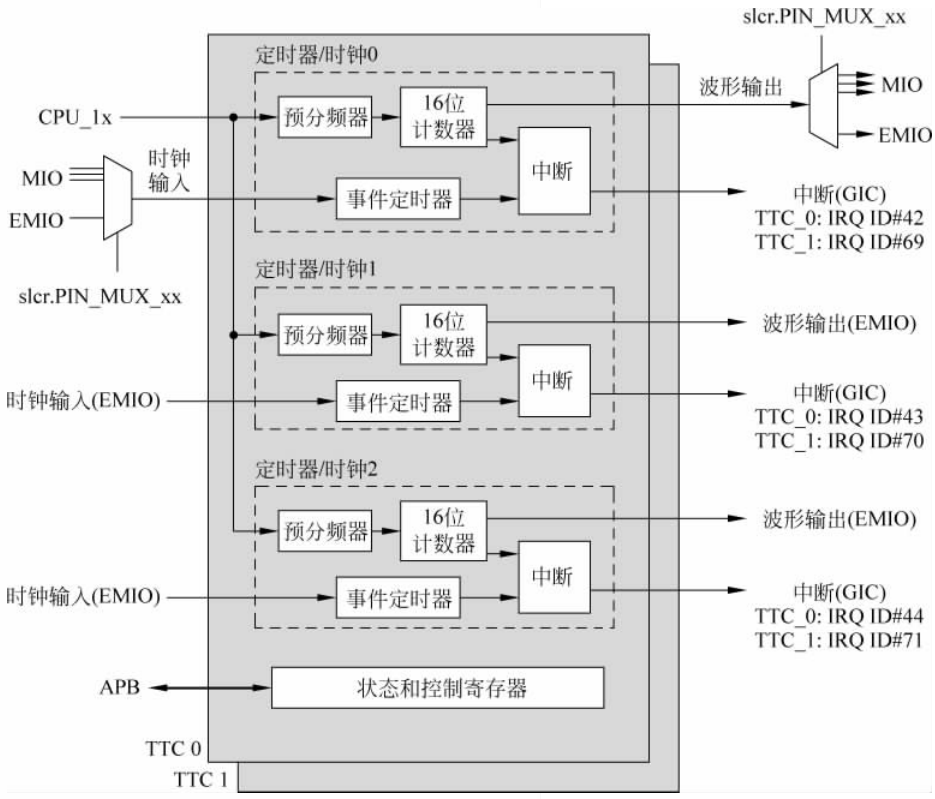

图 3.13　三重定时器/计数器块图结构

2. 三重定时器/计数器的功能

可以对每个预分频模块独立进行编程。这样预分频模块可以使用 PS 内部的总线时钟(CPU_1x)，或者来自 MIO 或者 PL 的外部时钟。对于一个外部时钟，SLCR 寄存器用于确定通过 MIO 或者来自 PL 的正确引脚输出。在应用到计数器前，将所选择时钟的频率降低到 2～65 536 分频范围。

计数器模块可以向上或者向下计数，并且能在一个给定的配置间隔内计数。它也将三个匹配寄存器和计数器的值进行比较，如果有一个匹配，则产生中断。

中断模块将各种类型的中断进行组合：计数器间隔、计数器匹配、计数器溢出、事件定时器溢出。中断模块能单独使能每个类型的中断。

每个计数器的模块能独立编程，使其工作在下面的两个模式：间隔模式或者溢出模式。

1）间隔模式

计数器在给定的0到间隔寄存器值之间，连续地递增或递减。计数的方向由计数器控制寄存器的DEC比特位确定。当计数器穿过零时，产生一个间隔中断。当计数器的值等于其中一个匹配寄存器的值时，产生相应的匹配中断。

2）溢出模式

计数器在0和0xFFFF之间，连续地递增或递减。计数的方向由计数器控制寄存器的DEC比特位确定。当计数器穿过零时，产生一个溢出中断。当计数器的值等于其中一个匹配寄存器的值时，产生相应的匹配中断。

事件定时器通过一个用户不可见的、由CPU_1x时钟驱动的16位内部计数器，进行操作：

（1）在外部脉冲的非计数周期，复位到0；

（2）在外部脉冲的计数周期，递增。

事件控制定时器寄存器用于控制内部定时器的行为：

（1）E_En比特：为0时，立即复位内部计数器到0，并且停止递增。

（2）E_Lo比特：指定外部脉冲的计数周期。

（3）E_Ov比特：指定如何管理内部计数器的溢出（在外部脉冲的计数周期）。

① 当0：溢出使得E_En为0。

② 当1：溢出使得内部计数器回卷和继续递增。

③ 当发生移出时，总是产生一个中断。

在外部脉冲计数周期结束时，使用非零值更新内部计数器的事件寄存器。因此，它表示了外部脉冲的宽度，用于测量CPU_1x的周期数。

如果内部计数器复位为零时，由于溢出，在外部脉冲的计数周期内，不更新事件寄存器，使其能保持最后一个非溢出操作原来的值。

表3.28给出了TTC的寄存器概述。

表3.28 TTC寄存器概述

功　能	名　字	概　述
时钟控制	时钟控制寄存器	控制预分频器，选择时钟输入，边沿
	计数器控制寄存器	使能定时器，设置操作模式，设置上/下计数，使能匹配，使能波形输出
状态	计数器值寄存器	返回当前计数器的值
计数器控制	间隔寄存器	设置间隔值
	匹配寄存器1 匹配寄存器2 匹配寄存器3	设置匹配值，总共3个
中断	中断寄存器	显示当前中断状态
	中断使能寄存器	使能中断
事件	事件控制定时器寄存器	使能事件定时器，停止定时器，设置相位
	事件寄存器	显示外部脉冲的宽度

3. 三重定时器/计数器的编程模型

1) 计数器的使能序列

(1)选择时钟输入源,设置预分频器的值(SLCR MIO_MUX_SEL 寄存器,TTC 时钟控制寄存器):在继续这一步前,确认禁止 TTC(TTC[Counter_Control_n[DIS]]=1)。

(2) 设置间隔值(间隔寄存器):这步可选。只用于间隔模式。

(3) 设置匹配值(匹配寄存器):这步可选。如果使能匹配模式,则设置匹配值。

(4) 使能中断(中断使能寄存器):这步可选。如果使能中断,则设置使能中断。

(5)使能/禁止波形输出,使能/禁止匹配,设置计数方向,设置模式,使能计数器(TTC 计数器控制寄存器):这步启动计数器。

2) 计数器停止序列

(1) 回读计数器控制寄存器的值。

(2) 设置 DIS 位为 1,而保持其他比特位的值。

(3) 写回到计数器控制寄存器。

3) 计数器重新启动序列

(1) 回读计数器控制寄存器的值。

(2) 设置 RST 位为 1,而保持其他比特位的值。

(3)写回到计数器控制寄存器。

4) 事件定时器使能序列

(1)选择外部脉冲源(SLCR MIO_MUX_SEL 寄存器):所选择的外部脉冲宽度由 CPU_1x 周期测量。

(2)设置溢出管理,选择外部脉冲电平,使能事件定时器(事件控制定时器寄存器):这步开始测量所选择电平的外部脉冲宽度。

(3)使能中断(中断使能寄存器):这步是可选的。如果使能中断,则设置使能中断。

(4)读测量的宽度(事件寄存器):当发生溢出时,返回的值是不正确的。

5) 中断清除和响应序列

读中断寄存器:当读的时候,清除中断寄存器所有位。

3.12.5 I/O 信号

表 3.29 给出了定时器 I/O 信号的列表。这里有两个三重定时器计数器(TTC0 和 TTC1)。每个 TTC 有三个接口信号集:时钟输入和波形输出,用于计数器/定时器 0、1 和 2。

对于每个三重定时器来说,来自计数器/定时器 0 的信号能使用 MIO_PIN 寄存器连接到 MIO。如果 MIO_PIN 寄存器没有选择时钟输入或者波形输出,则默认信号连接到 EMIO。

只能通过 EMIO 使用,用于计数器/定时器 1 和 2 的信号。

表 3.30 给出了看门狗定时器的 I/O 信号。

表 3.29 TTC I/O 信号

TTC	定时器信号	I/O	MIO 引脚	EMIO 信号	控制器默认输入值
TTC0	计数器/定时器 0 时钟输入	I	19,31,43	EMIOTTC0CLKI0	0
	计数器/定时器 0 波形输出	O	18,30,42	EMIOTTC0WAVEO0	~
	计数器/定时器 1 时钟输入	I	N/A	EMIOTTC0CLKI1	0
	计数器/定时器 1 波形输出	O	N/A	EMIOTTC0WAVEO1	~
	计数器/定时器 2 时钟输入	I	N/A	EMIOTTC0CLKI2	0
	计数器/定时器 2 波形输出	O	N/A	EMIOTTC0WAVEO2	~
TTC1	计数器/定时器 0 时钟输入	I	17,29,41	EMIOTTC1CLKI0	0
	计数器/定时器 0 波形输出	O	16,28,40	EMIOTTC1WAVEO0	~
	计数器/定时器 1 时钟输入	I	N/A	EMIOTTC1CLKI1	0
	计数器/定时器 1 波形输出	O	N/A	EMIOTTC1WAVEO1	~
	计数器/定时器 2 时钟输入	I	N/A	EMIOTTC1CLKI2	0
	计数器/定时器 2 波形输出	O	N/A	EMIOTTC1WAVEO2	~

表 3.30 看门狗定时器的 I/O 信号

SWDT 信号	I/O	MIO 引脚	EMIO 信号	控制器默认输入值
时钟输入	I	14,26,38,50,52	EMIOWDTCLKI	0
复位输出	O	15,27,39,51,53	EMIOWDTRSTO	~

思考题 3-17：Zynq-7000 内提供了哪些类型的定时器？这些类型的定时器实现什么功能？

3.13 DMA 控制器

DMA 控制器(DMA Controller,DMAC)使用一个 AXI 主接口,在系统存储器和外设之间的传输数据。

3.13.1 DMA 控制器结构及特性

DMAC 包含一个小的指令集,它为指定的 DMAC 操作提供了一个灵活的方法。这比基于固定能力的链接列表项(Linked-List Item,LLI)的 DMA 控制器来说,提供了更大的灵活性。程序代码保存在系统存储器空间范围内,DMAC 使用它的 AXI 主设备接口访问程序代码。DMAC 能配置最多 8 个通道,每个通道能支持一个单独并发的 DMA 操作线程。DMAC 有一个 DMA 管理器,用于初始化 DMA 通道线程。当一个 DMA 通道线程执行一个加载或者保存指令时,DMAC 将指令添加到相关的读或者写队列中。

在 AXI 总线上发布命令时,DMAC 使用这些队列作为一个指令存储缓冲区。在一个 DMA 传输过程中,DMAC 也包含一个多通道先进先出队列(Multichannel Fisrt-in-First-Out,MFIFO)数据缓冲区,用于保存读或者写数据。对于一个程序,它看上去好像是一个可变深度的 FIFO 集。每个通道都有一个 FIFO 集,其限制是所有 FIFO 总的深度不能超过 MFIFO 的大小。

DMAC 可以在没有 CPU 干预的情况下,在存储器之间移动数据。源和目的存储器

应该是 PS 或者 PL 系统内的任何存储器资源。

对带有 PS 存储器的传输流量控制是使用 AXI 互联。带有 PL 外设的访问可以使用 AXI 流控制或者 DMAC 的 PL 外设请求接口。没有外设请求接口直接指向 PS I/O 外设。对于 PL 外设的 AXI 交易,运行在 CPU 上的软件使用中断或轮询的可编程 I/O 方法。

控制器有两套控制和状态寄存器。一套在安全模式下访问,另一套在非安全模式下访问。软件通过控制器的 32 位 APB 从接口访问这些寄存器。整个控制器工作在安全或者非安全模式下。在一个通道中,不存在安全和非安全的混合模式。安全配置的改变由 slcr 寄存器控制。并且,要求复位控制器使得这些改变发生作用。

DMAC 具有以下特性。

(1) DMA 引擎处理器带有一个灵活的指令集,用于 DMA 传输:①灵活地分散-聚集存储器传输;②充分地控制用于源和目的的寻址;③定义 AXI 交易属性;④管理字节流。

(2) 支持 8 个缓存行,每个缓存行是 4 个深度。

(3) 支持 8 个并发的 DMA 通道:①允许并行执行多个线程;②发布命令,最多 8 个读和 8 个写 AXI 交易。

(4) 8 个到 PS 中断控制器和 PL 的中断。

(5) 在 DMA 引擎程序代码内的 8 个事件。

(6) 用于数据缓冲的 128(64 位)字 MFIFO。在一个传输过程中控制器读或写 MFIFO。

(7) 安全性:①专用的 APB 从接口,用于安全寄存器的访问;②将整个控制器配置为安全的或者不安全的。

(8) 存储器-存储器的 DMA 传输。

(9) 四个 PL 外设请求接口,用于控制进入或者从 PL 逻辑输出的流。每个接口支持最多 4 个活动的请求。

图 3.14 给出了 DMA 控制器的系统视图。

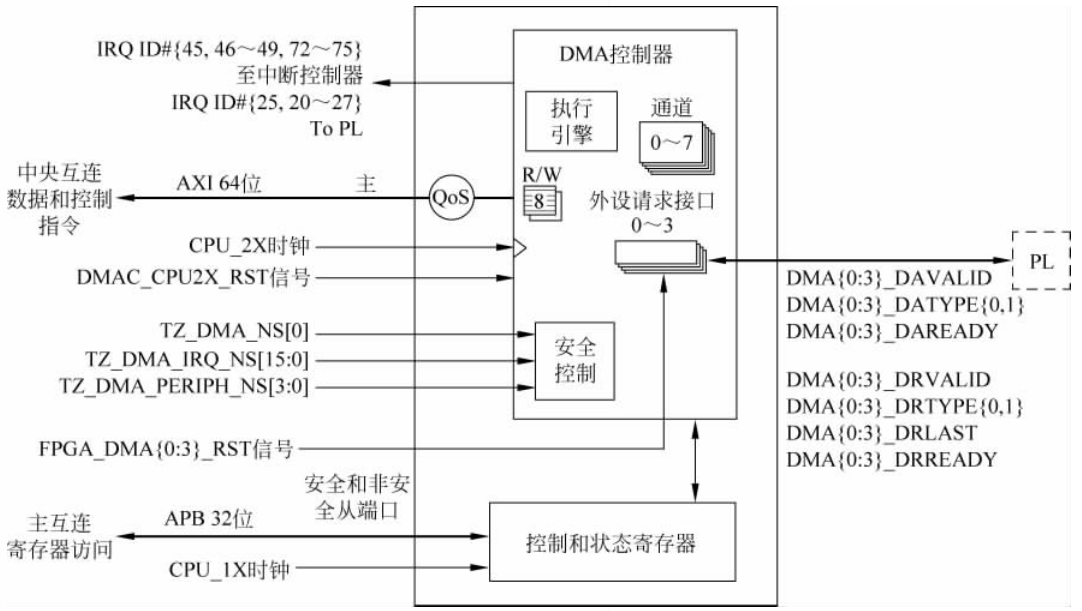

图 3.14 DMA 控制器系统视图

图 3.15 给出了 DMA 控制器的块图。下面将介绍该 DMA 控制器的内部结构。

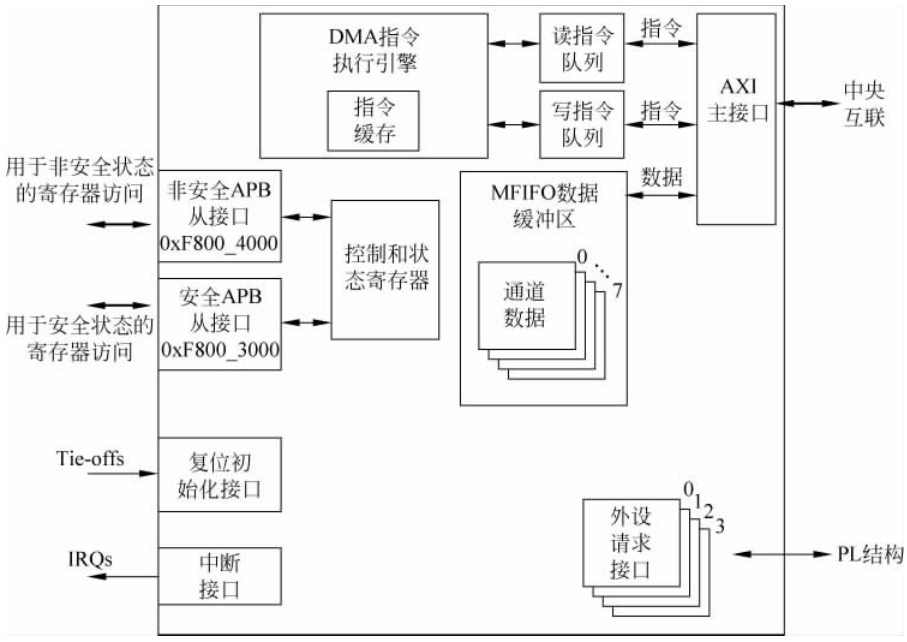

图 3.15　DMA 控制器块图

1. DMA 指令执行引擎(操作状态)

DMAC 包含一个指令处理模块,用来处理用于控制一个 DMA 传输的程序代码。DMAC 为每个线程都保留一个独立的状态机。

2. 指令缓存

DMAC 暂时将指令保存在缓存中。当一个线程请求来自一个地址的指令时,缓存执行查找。如果缓存命中,则缓存立即提供数据。否则,停止线程,DMAC 使用 AXI 接口执行一个来自系统存储器的缓存行填充。如果指令大于四个字节或者跨过缓存行的结尾,则执行多个缓存访问来取出指令。

注:当正在进行缓冲行填充时,DMAC 使能其他线程访问缓存。但是,如果发生其他缓存缺失,停止流水线,直到完成第一个缓存行的填充。

3. 读/写指令队列

当一个 DMA 通道线程执行一个加载或者保存指令时,DMAC 添加指令到相关的读或者写队列中。在 AXI 中央互连上发布指令前,DMAC 使用这些队列作为保存一个指令的缓冲区。

4. 多通道数据 FIFO

在 DMA 传输期间,如果 DMAC 执行读或写操作,则 DMAC 使用多通道先进先出 MFIFO 数据缓冲区保存数据。

5. 用于取指和 DMA 传输的 AXI 主接口

程序代码保存在系统存储器的区域内，这个 AXI 主接口用于访问 DMAC。AXI 主接口也可以使能 DMAC，将数据从一个源 AXI 从设备传输到一个目的 AXI 从设备。

6. 用于访问寄存器的 APB 从接口

DMAC 提供了这些 APB 接口：
(1) 非安全的 APB 从接口；
(2) 安全的 APB 从接口。
APB 从接口将 DMAC 连接到 APB。并且，微处理器能够访问这些寄存器。

7. 控制和状态寄存器

使用这些寄存器，一个微处理器具有以下功能：
(1) 访问 DMA 管理器的状态；
(2) 访问 DMA 通道线程的状态；
(3) 使能或者清除中断；
(4) 使能事件；
(5) 发布一个指令，用于 DMAC 执行。

8. PL 外设 DMA 请求接口

外设请求接口支持 PL 内，有 DMA 能力的外设连接。每个外设请求接口和其他外设请求接口之间是异步的，并且和 DMA 本身也是异步的。

9. 中断接口

中断接口用于事件和中断控制器之间高效地通信。

10. 复位初始化接口

当退出复位时，这个接口使能用户初始化 DMAC 的操作状态。当 DMAC 从复位退出时：

(1) 它读取 SLCR 寄存器 TZ_DMA_NS 的状态，用来设置 DMA 管理器的安全性；

(2) 它读取 SLCR 寄存器 TZ_DMA_PERIPH_NS 的状态，用来将每个外设请求接口分配到安全状态；

(3) 它读取 SLCR 寄存器 TZ_DMA_IRQ_NS 的状态，用来将每一个 irq[x]信号分配到安全状态；

(4) 它等待来自其中一个 APB 接口的指令。

3.13.2 DMA 控制器功能

所有的 DMA 交易均使用 AXI 主接口移动数据。如果 PL 内一个有 DMA 能力的外设参与到交易中，使用四个外设请求接口中的一个接口，用于控制 AXI 上的数据流，以避

免上溢或者下溢的问题。如图 3.16 所示，这里有两个类别的 DMA 交易：

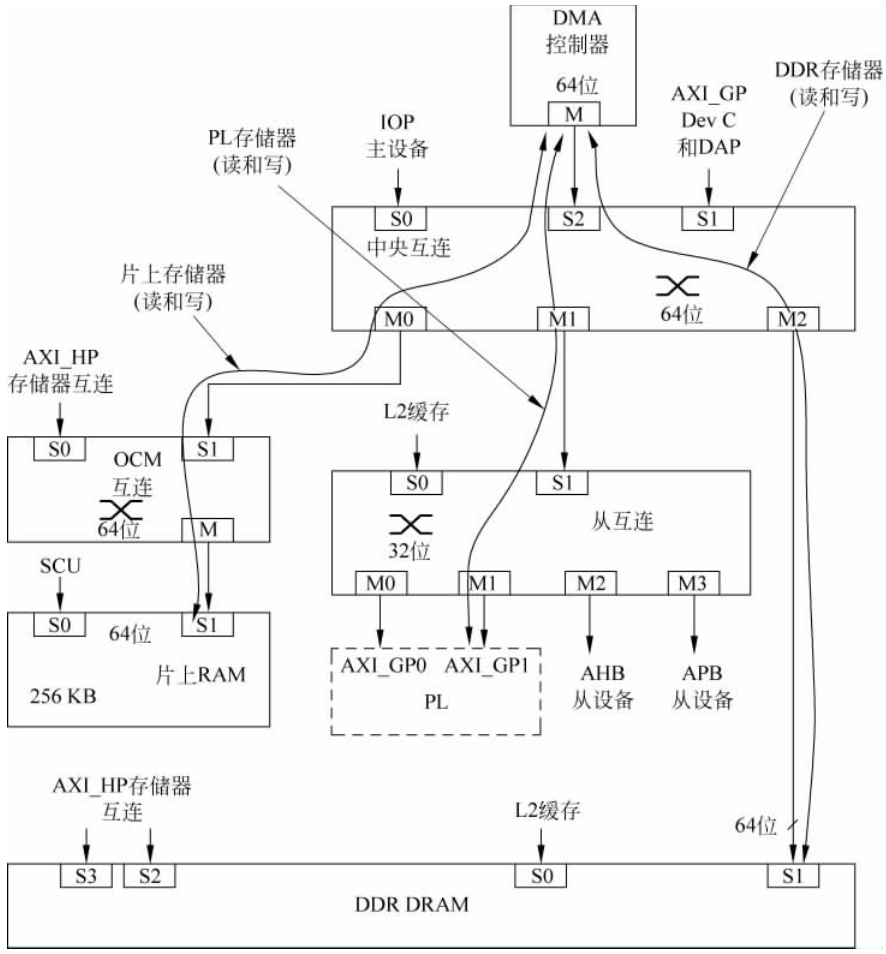

图 3.16　DMAC 读/写 DDR，OCM 和 PL 外设

(1) 存储器到存储器交易；

(2) 存储器和 PL 之间的交易。

当 DMAC 工作在实时情况下，用户只能发布下面有限的指令子集：

(1) DMAG：使用用户指定的一个 DMA 通道，开始 DMA 交易。

(2) DMASEV：使用用户指定的一个事件号，发送一个事件或发生中断的信号。

(3) DMAKILL：终止一个线程。

必须使用合适的 APB 接口。这取决于 DMA 管理器在对寄存器 TZ_DMA_NS 初始化时所设置的安全状态。例如，如果 DMA 管理器处于安全状态时，必须使用在安全 APB 接口的指令，否则 DMAC 将忽略这些指令。当 DMA 管理器在非安全状态时，建议使用非安全的 APB 接口，用于启动或者重新启动一个 DMA 通道。安全 APB 接口可以工作在非安全模式下。

在使用调试指令寄存器或者 DBGCMD 寄存器发布指令前，必须读 DBGSTATUS 寄存器，以确保调试器处于空闲状态。否则，DMA 管理器将忽略这些指令。

当 DMA 管理器从 APB 从接口接收到一条指令时，需要一些时钟周期才能处理这条

指令。例如，此时流水线正忙于处理其他指令。

在发布 DMAGO 指令前，系统存储器必须包含一个合理的程序，该程序用于执行 DMA 通道线程。由 DMAGO 指定通道线程的开始地址。

1. 存储器-存储器交易

控制器使用 AXI 中央互联开关上的一个主设备访问系统内的存储器，包括：

(1) OCM；

(2) DDR。

通过相同的 AXI 中央互联，控制器也能访问绝大多数的外设子系统。如果可以将一个目标外设看作存储器映射的一个区域(或者存储器的端口位置)，并且不需要使用 FIFO 进行流量控制，此时 DMAC 就可以用于读/写目标外设。典型的例子如下：

(1) 线性地址空间的 QSPI；

(2) NOR Flash；

(3) NAND Flash。

2. 存储器-PL 外设交易

绝大多数的外设允许通过 FIFO 进行传输数据。必须管理这些 FIFO，以避免上溢和下溢条件。由于这个原因，四个指定的外设请求接口可用于将 DMAC 连接到 PL 内的 DMA 设备中。这些接口中的每一个都能分配到任意一个 DMA 通道。

配置 DMAC，使得其可以为每一个外设接口接受最多四个活动的请求。一个活动的请求是指：DMAC 还没有开始所要求的 AXI 数据传输。DMAC 有一个请求 FIFO，用于每个外设接口，它用来捕获来自一个外设的请求。当一个请求 FIFO 填满时，DMAC 设置相应的 DMA{3:0}_DRREADY 为低，用来发信号表示 DMAC 不能再接受任何来自外设的请求。

注：在 PS 内没有外设请求接口直接指向 I/O 外设(IOP)。这需要处理器的干预，以避免在目标 PS 外设内 FIFO 出现上溢或者下溢的条件。

这里有两种不同的方法来管理 DMAC 和外设之间数据流的数量。

(1) PL 外设长度管理：在一个 DMA 周期内，PL 外设控制所包含数据的数量。

(2) DMAC 长度管理：在一个 DMA 周期内，DMAC 控制所包含数据的数量。

3. 外设请求接口

图 3.17 给出了外设请求接口，它由外设请求总线和一个 DAMC 响应总线构成。

1) 使用的前缀

(1) DR：外设请求总线。

(2) DA：DMAC 响应总线。

所有的总线使用 AXI 协议中所描述的有效-准备(valid-ready)握手信号。

外设使用 DMA{3:0}_DRTYPE[1:0]寄存器：

(1) 请求一个单个 AXI 交易；

(2) 请求一个 AXI 猝发交易；

(3) 响应一个刷新请求。

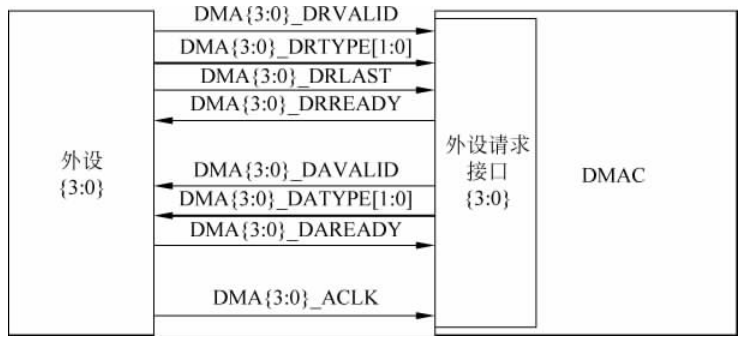

图 3.17　在外设请求接口上的请求和响应总线

DMAC 使用 DMA{3:0}_DATYPE[1:0]：

(1) 当完成一个请求的单次传输后，发信号；

(2) 当完成一个请求的猝发传输后，发信号；

(3) 发布一个刷新请求。

PL 外设使用 DMA{3:0}_DRLAST：当最后一个 DMA 传输开始时，发信号给 DMAC。

2) 握手规则

DMAC 使用表 3.31 给出的 DMA 握手规则。当一个 DMA 通道线程是活动时，即不在停止状态。

表 3.31　握手规则

规则	描述(1)
1	在任何 DMA{3:0}_ACLK 周期，DMA{3:0}_DRVALID 能从低到高变化；但是当 DMA{3:0}_DRREADY 为高时，只能从高到低变化
2	只有下面情况，DMA{3:0}_DRTYPE 才能发生变化： (1) DMA{3:0}_DRREADY 为高 (2) DMA{3:0}_DRVALID 为低
3	只有下面情况，DMA{3:0}_DRLAST 才能发生变化： (1) DMA{3:0}_DRREADY 为高 (2) DMA{3:0}_DRVALID 为低时
4	在任何 DMA{3:0}_ACLK 周期，DMA{3:0}_DAVALID 能从低到高变化；但是当 DMA{3:0}_DAREADY 为高时，只能从高到低变化
5	只有下面情况，DMA{3:0}_DATYPE 才能发生变化： (1) DMA{3:0}_DAREADY 为高 (2) DMA{3:0}_DAVALID 为低时

注：所有的信号与 DMA{3:0}_ACLK 同步。

3) 将 PL 外设接口映射到 DMA 通道

通过 DMAC，使得软件能将一个外设请求接口分配给任何一个 DMA 通道。当一个 DMA 通道线程执行 DMAWFP 时，外设[4:0]位域所编程的值表明和该 DMA 通道相关的设备。

4）PL 外设长度管理

外设请求接口用来使能外设，控制一个 DMA 周期所包含数据的数量。而 DMAC 不需要知道所包含需要传输的数据个数。通过使用下面的信号设置，实现 PL 外设控制 AXI 交易。

（1）DMA{3:0}_DRTYPE[1:0]：选择单个或者猝发传输。

（2）DMA{3:0}_DRLAST：在当前传输序列中，当它开始最后的请求时，通知 DMAC。

当 DMAC 执行一个 DMAWFP 指令时，它停止执行线程。并且，等待外设发送一个请求。当外设发送请求时，DMAC 根据下面的信号设置请求标志的状态：

（1）DMA{3:0}_DRTYPE[1:0]。DMAC 设置 request_type 标志：①＝b00，request_type{3:0}＝Single；②＝b01，request_type{3:0}＝Burst。

（2）DMA{3:0}_DRLAST。DMAC 设置 request_last 标志：①＝0，request_last{3:0}＝0。②＝1，request_last{3:0}＝1。

如果 DMAC 执行一个 DMAWFP 单个或者 DMAWFP 猝发指令，则 DMAC 设置：

（1）request_type{3:0}标志为 Single 或者 Burst；

（2）request_last{3:0}标志为 0。

DMALPFE 是一个汇编器命令，用来强制将相关 DMALPEND 指令的 nf 比特位设置为 0。这用来创建一个程序循环，不使用循环计数器终止循环。当 request_last 标志设置为 1 时，DMAC 退出循环。

（1）根据 request_type 和 request_last 标志的状态，DMAC 有条件的执行下面的指令：

```
DMALD,DMAST,DMALPEND
```

（2）当这些指令使用可选的 B|S 后缀时，如果与 request_type 标志不匹配，则 DMAC 执行 DMANOP：

```
DMALDP<B|S>,DMASTP<B|S>
```

（3）如果 request_type 标志不匹配 B|S 后缀时，DMAC 执行一个 DMANOP 指令：

```
DMALPEND
```

当 nf 比特为零时，如果设置 request_last 标志，则 DMAC 执行一个 DMANOP 指令。

当 DMAC 接收到一个猝发请求（即 DMA{3:0}_DRTYPE[1:0]＝b01）时，如果要求 DMAC 发布一个猝发传输，则使用 DMALDB、DMALDPB、DMASTB 和 DMASTPB 指令。CCRn 寄存器的值控制 DMAC 传输数据的数量。

当 DMAC 接收到一个单个请求（即 DMA{3:0}_DRTYPE[1:0]＝b00）时，如果要求 DMAC 发布一个单个传输，则使用 DMALDS、DMALDPS、DMASTS 和 DMASTPS 指令。DMAC 忽略 CCRn 寄存器中 src_burst_len 和 dst_burst_len 域的值。并且，设置 arlen[3:0]和 awlen[3:0]总线为 0x0。

5）DMAC 长度管理

通过 DMAC 长度管理，DMAC 控制传输数据的总个数。使用外设请求接口，当按照所要求的一个方向传输数据时，一个外设通知 DMAC。DMA 通道线程控制 DMAC 如何

响应外设请求。

对于DMAC长度管理，有下面的约束：

(1) 来自一个外设的用于所有单个请求的总的数据量，必须小于来自那个外设的一个猝发请求的数据量。

(2) CCRn寄存器控制用于一个猝发请求和一个单个请求所传输数据的个数，ARM推荐：当正在处理通道n的传输时，不要更新CCRn寄存器。

(3) 当外设发送一个猝发请求后，外设不得发送一个单个请求，直到DMAC响应完成了猝发请求为止。

当要求停止程序线程，就要使用DMAWFP单个指令，直到外设请求接口接收到任何请求类型。如果在请求FIFO中的头部入口的请求类型是：

(1) 单个：DMAC从FIFO弹出入口，继续执行程序。

(2) 猝发：DMAC在FIFO中留下入口，继续执行程序。

注：猝发请求入口保留在请求FIFO中，直到DMAC执行DMAWFP猝发指令或者DMAFLUSTP指令。

当要求停止程序线程，就要使用DMAWFP猝发指令，直到外设请求接口接收到一个猝发请求类型。如果在请求FIFO中的头部入口的请求类型是：

(1) 单个：DMAC从FIFO移除入口，程序的执行保持停止。

(2) 猝发：DMAC从FIFO弹出入口，继续程序的执行。

当完成一个AXI读交易，要求DMAC发送一个响应给PL外设时，应该使用DMALDP指令。类似地，当完成一个AXI写交易，要求DMAC发送一个响应给PL外设时，应该使用DMASTP指令。DMAC使用DMA{3:0}_DATYPE[1:0]总线，发送一个到PL外设{3:0}的交易响应。

当rvaild和rlast为高时，DAMC为读交易发送响应；当bvalid为高时，DMAC为写交易发送一个响应。如果系统能缓冲AXI写交易时，DAMC可能给外设发送一个响应。但是，到终端目的写数据仍然在进行。

DMAFLUSHP指令用于复位外设请求接口的请求FIFO命令。当DMAC执行DMAFLUSHP时，它忽略外设请求，直到外设响应刷新请求。这使得DMAC和外设互相同步。

6) 外设请求接口时序

图3.18给出了当一个外设请求一个猝发传输时，使用前面介绍的握手，外设请求接口的操作的例子。

(1) T1周期：DMAC检测到一个用于猝发传输的请求。

(2) T2～T6周期：DMAC执行一个猝发传输。

(3) T7：DMAC设置DMA{3:0}_DAVALID为高，设置DMA{3:0}_DATYPE[1:0]表示完成猝发传输。

4. 多通道数据FIFO

MFIFO是一个由当前所有活动通道共享的，基于先进先出的共享资源。对于一个程序，它看上去是深度可变的并行FIFO的集合。每个通道都有一个FIFO。但是，所有FIFO的总的深度不能超过MFIFO的大小。DMAC最大的MFIFO深度为128(64位)

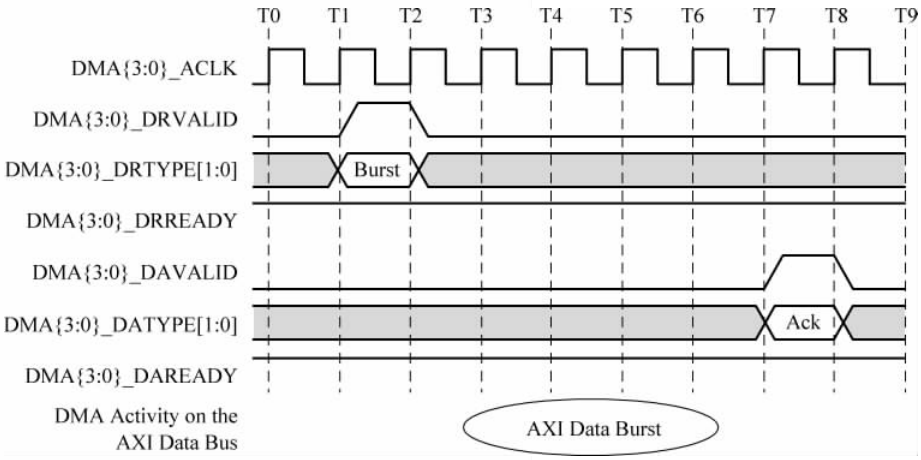

图 3.18　猝发请求信号

的字。

控制器能将源数据重新对齐到目的。例如，从地址 0x103 读一个字，写到地址 0x205 时，DMAC 将数据移动两个字节。由目的地址和传输的特征来确定 MFIFO 中数据的保存和封装。

当一个程序指定将要执行一个到目的递增交易时，DMAC 将数据打包到 MFIFO，以使用最少的 MFIFO 入口。例如，当 DMAC 有一个 64 位的 AXI 数据总线，程序使用 0x100 的源地址和 0x200 的目的地址时，DMAC 将两个 32 位的字打包到 MFIFO 的单个入口。

在某些条件下，要求保存来自源数据的入口个数，不是简单地通过总的源数据除以 MFIFO 的宽度的计算得到。当下面情况发生时，计算入口的个数并不简单：

(1) 源地址没有对齐 AXI 总线宽度；

(2) 目的地址没有对齐 AXI 总线宽度；

(3) 到一个固定目的的交易，即非递增的地址。

DMALD 和 DMAST 指令指明了将要执行的一个 AXI 总线交易。根据 CCRn 寄存器编程的值和交易的地址，决定一个 AXI 总线交易所要传输的数据的个数。

5. 事件和中断

DMAC 支持 16 个事件。这些事件中的前 8 个可以用于中断信号 IRQs[7:0]。8 个中断的每一个可以同时输出到 PS 中断控制器和 PL。表 3.32 给出了 DMAC IRQ＃和系统 IRQ＃之间的映射。

表 3.32　事件/中断源

DMAC IRQ＃	系统 IRQ＃(到处理器)	系统 IRQ＃(到 PL)	DMA 引擎事件＃
0～3	46～49	20～23	0～3
4～7	72～75	24～27	4～7
8～15	N/A	N/A	8～15

当 DMAC 执行一个 DMASEV 指令时，它修改用户指定的事件/中断：

(1) 如果 INTEN 寄存器将事件/中断源作为一个事件时，DMAC 为指定的事件/中断源产生一个事件。当 DMAC 为相同的事件-中断源执行一个 DMAWFE 指令后，清除事件。

(2) 如果 INTEN 寄存器将事件/中断源作为一个中断时，DMAC 将 irq<事件号>设置为高。其中，事件号为指定事件源的编号。用户通过写 INTCLR 寄存器来清除中断。

6. 退出

通过 IRQ#45，给 CPU 发送退出；以及通过 IRQP2F[28]信号，给 PL 外设发送退出。表 3.33 给出了所有可能引起退出的原因。表 3.34 给出了当一个退出条件后，DMAC 的行为。当发生退出条件时，DMAC 所采取的行为取决于线程的类型。表 3.35 给出了当接收到退出条件时，处理器或者 PL 外设必须产生的行为。

表 3.33 退出类型和条件

退出类型	条件
精确的：DMAC 用创建退出指令的地址更新 PC 寄存器 注意，当 DMAC 发出精确的退出后，不执行触发退出的指令；取而代之的是 DMAC 执行一个 DMANOP	**通道控制寄存器的安全冲突**：一个非安全状态下的一个 DMA 通道线程，尝试对通道控制寄存器编程和产生一个安全的 AXI 交易
	事件的安全冲突：在一个非安全状态下的一个 DMA 通道线程执行 DMAWFE 或者 DMASEV 指令，用于设置为安全的事件。SLCR 寄存器的 TZ_DMA_IRQ_NS 控制着一个事件的安全状态
	外设请求接口的安全冲突：在一个非安全状态下的一个 DMA 通道线程执行 DMAWFP、DMALDP、DMASTP 或者 DMAFLUSHP 指令，用于设置为安全的外设请求接口，SLCR 寄存器的 TZ_DMA_PERIPH_NS 控制着一个外设请求接口的安全状态
	DMAGO 的安全冲突：在一个非安全状态下，DMA 管理器执行 DMAGO，尝试启动一个安全的 DMA 通道线程
	AXI 主接口的错误：当执行一个取指时，在 AXI 主接口上 DMAC 接收到一个 ERROR 响应。比如尝试访问保留的存储器空间
	执行引擎的错误：一个线程执行一个未定义的指令或者执行带有一个无效操作数的用于配置 DMAC 的指令
不精确的：PC 寄存器可能包含没有引起退出发生指令的地址	**数据加载错误**：当执行一个数据加载时，在 AXI 主接口上 DMAC 接收到一个 ERROR 响应
	数据存储错误：当执行一个数据存储时，在 AXI 主接口上 DMAC 接收到一个 ERROR 响应
	MFIFO 错误：一个 DMA 线程执行一个 DMALD，但是 MFIFO 太小以至于不能保存数据；或者执行 DMAST，MFIFO 没有足够的数据用来完成数据的传输
	看门狗退出：当正在执行一个或者多个 DMA 通道程序时，MFIFO 太小以至于不能满足 DMA 程序保存的要求时，锁定 DMAC DMAC 包含逻辑，用来阻止其一直保持在不能完成一个 DMA 传输的状态当下面的所有条件发生时，DMAC 检测锁定：①加载队列为空；②保存队列为空；③由于 MFIFO 没有足够的空闲空间或者其他通道拥有加载锁，阻止所有的运行通道执行一个 DMALD 指令 当 DMAC 检测到一个锁定时，它发一个中断信号，也能放弃当前使用的通道。DMAC 的行为取决于 WD 寄存器中 wd_irq_only 比特位的状态

表 3.34　退出处理

线 程 类 型	DMAC 行为
通道线程	设置 IRQ#45 和 IRQ#28 为高
	停止执行用于 DMA 通道的指令
	用于 DMA 通道的所有缓存入口无效
	更新通道程序计数器的寄存器，用来包含退出指令的地址，该地址是精确类型退出的地址
	不产生对任何保留在读队列和写队列指令的 AXI 访问
	允许完成当前活动的 AXI 交易
DMA 管理器	设置 IRQ#45 中断和 IRQ#28 信号为高

表 3.35　线程终止

读 DMA 管理器寄存器的故障状态，用来确定 DMA 管理器所处的故障状态，以及确定引起退出的原因
读 DMA 通道寄存器的故障状态，用来确定 DMA 通道所处的故障状态，以及确定引起退出的原因
编程调试指令- 0 寄存器
写调试命令寄存器

7. 安全性

当 DMAC 从复位退出时，复位状态初始化接口信号配置安全性，用于：

(1) DMA 管理器(SLCR 寄存器 TZ_DMA_NS)；

(2) 事件/中断源(SLCR 寄存器 TZ_DMA_IRQ_NS)；

(3) PL 外设请求接口(SLCR 寄存器 TZ_DMA_PERIPH_NS)。

当 DMA 管理器为 DMA 执行 DMAGO 指令时，它设置 ns 比特位来设置通道的安全状态。通道的状态由通道状态寄存器内的动态非安全比特 CNS 提供。

表 3.36 给出了对于 ARM 的命名，在本章中的命名规则。表 3.37 和表 3.38 分别给出了用于 DMA 管理器和 DMA 通道线程的安全性。

表 3.36　安全用法命名规则

ARM 名字	XILINX 名字	描　　述
DNS	TZ _ DMA _ NS 内的 DMAC_NS	当 DMAC 从复位退出时，这个信号控制 DMA 管理器的安全状态 0：DMA 管理器运行在安全状态下 1：DMA 管理器运行在非安全状态下
INS	TZ_DMA_IRQ_NS 内的 DMAC_IRQ_NS<x>	当 DMAC 从复位退出时，这个信号控制事件/中断的安全性 0：DMA 中断/事件比特位在安全状态下 1：DMA 中断/事件比特位在非安全状态下
PNS	TZ_DMA_PERIPH_NS 内的 DMAC_PERIPH_NS<x>	当 DMAC 从复位退出时，这个信号控制外设请求接口安全性 0：DMA 外设请求接口处于安全状态下 1：DMA 外设请求接口处于非安全状态下
ns	DMAGO 指令内的 ns	DMAGO 指令的 1 比特 0：在安全状态下启动 DMA 通道线程 1：在非安全状态下启动 DMA 通道线程
CNS	CSR<x>内的 CNS	通道状态寄存器的 CNS 比特位为每个 DMA 通道的安全状态提供 0：DMA 通道线程工作在安全状态下 1：DMA 通道线程工作在非安全状态下

表 3.37 用于 DMA 管理器的安全性

	DNS	指令	PNS	INS	描 述
DMA 管理器	0	DMAGO	0	—	使用安全 APB 接口，发布指令，在安全状态下，启动 DMA 通道线程(CNS=0)
			1	—	使用安全 APB 接口，发布指令，在非安全状态下，启动 DMA 通道线程(CNS=1)
		DMASEV	—	×	使用安全 APB 接口，发布指令，它发出合适的事件信号，这和 INS 位无关
	1	DMAGO	0	—	使用非安全 APB 接口，发布指令，退出
			1	—	使用非安全 APB 接口，发布指令，DMA 通道线程处于非安全状态(CNS=1)
		DMASEV	—	0	使用非安全 APB 接口，发布指令，退出
			—	1	使用非安全 APB 接口，发布指令，它发出合适的事件信号

表 3.38 用于 DAM 通道线程的安全性

	DNS	指令	PNS	INS	描 述
DMA 通道线程	0	DMAWFE	—	×	出现事件，继续执行，不考虑 INS 比特
		DMASEV	—	×	给合适的事件发信号，不考虑 INS 比特
		DMAWFP	×	—	出现外设请求，继续执行，不考虑 PNS 比特
		DMALP，DMASTP	×	—	发消息给 PL 外设，告诉外设，DMA 传输的最后一个 AXI 交易已经完成，不考虑 PNS 比特
		DMAFLUSH	×	—	清除外设的状态，给外设发送一个消息，重新发送它的级别状态，不考虑 PNS 比特
	1	DMAWFE	—	0	退出
			—	1	出现事件，继续执行
		DMASEV	—	0	退出
			—	1	给合适的事件发信号
		DMAWFP	0	—	退出
			1	—	出现外设请求，继续执行
		DMALP，DMASTP	0	—	退出
			1	—	给外设发消息，告诉 DMA 传输的最后一个 AXI 交易已经完成
		DMAFLUSHP	0	—	退出
			1	—	它只清除外设的状态，发送消息给外设，重新发送它的级状态

3.13.3 外部信号

1. 外设请求接口

外设请求接口支持连接具有 DMA 能力的外设。在无须处理器干预的情况下，可以进行存储器-外设和外设到存储器的 DMA 传输。这些外设必须在 PL 内，连接到 M_AXI_GP 接口。所有的外设请求接口信号和各自的时钟同步。表 3.39 给出了 PL 外设请求接口信号线。

表 3.39 PL 外设请求接口信号

类 型	I/O	名 字	描 述
时钟	I	DMA{3:0}_ACLK	用于 DMA 请求传输的时钟
DMA 请求	I	DMA{3:0}_DRVALID	当外设提供有效的控制信息时,指示 0:没有可用的控制信息 1:DMA{3:0}_DRTYPE[1:0]和 DMA{3:0}_DRLAST 包含用于 DMAC 的有效信息
	I	DMA{3:0}_DRLAST	指示外设正在发送用于当前 DMA 传输的最后的 AXI 数据交易 0:最后一个数据请求不在进行中 1:最后一个数据请求正在进行中 注意,当 DMA{3:0}_DRTYPE[1:0]=00 或者 01 时,DMAC 只使用这个信号
	I	DMA{3:0}_DRTYPE[1:0]	指示一个响应或者请求的类型,外设发信号 00:单个级请求 01:猝发级请求 10:响应一个刷新请求,这个请求为 DAMC 请求 11:保留
	O	DMA{3:0}_DRREADY	指示 DMAC 是否能接收信息,这个信息由外设通过 DMA{3:0}_DRTYPE[1:0]提供 0:DMAC 没有准备好 1:DMAC 准备好
DMA 响应	O	DMA{3:0}_DAVALID	当 DMAC 提供有效的控制信息时,指示 0:没有可用的控制信息 1:DMA{3:0}_DATYPE[1:0]包含用于外设的有效信息
	I	DMA{3:0}_DAREADY	指示外设是否能接收信息,这个信息由 DMAC 通过 DMA{3:0}_DATYPE[1:0]提供 0:外设没有准备好 1:外设准备好
	I	DMA{3:0}_DATYPE[1:0]	指示一个响应或者请求的类型,DMAC 发信号 00:DMAC 已经完成了单个 AXI 交易 01:DMAC 已经完成了 AXI 猝发交易 10:DMAC 请求一个外设执行一个刷新请求 11:保留

2. AXI 主接口

DMAC 包含一个 AXI 主接口,使得能够从一个源 AXI 从接口将数据传输到一个目的 AXI 从接口。

3. 复位初始化接口

表 3.40 给出了用于编程 DMAC 安全状态的配合信号。根据复位后 SLCR 寄存器的状态,将 DMA 配置成安全或者非安全模式。

表 3.40 DMAC 初始化信号

名　字	类型	源	描　述
boot_manager_ns	输入	SLCR 寄存器 TZ_DMA_NS	当 DMAC 从复位退出时，控制 DMA 管理器的安全状态 0：给 DMA 管理器分配安全状态 1：给 DMA 管理器分配非安全状态
boot_irq_ns[15:0]	输入	SLCR 寄存器 TZ_DMA_IRQ_NS	当 DMAC 从复位退出时，控制事件-中断源的安全状态 (1) boot_irq_ns[x]为低时，给事件＜x＞或 irq＜x＞分配安全状态 (2) boot_irq_ns[x]为高时，给事件＜x＞或 irq＜x＞分配非安全状态
boot_periph_ns[3:0]	输入	SLCR 寄存器 TZ_DMA_PERIPH_NS	当 DMAC 从复位退出时，控制外设请求接口的安全状态 (1) boot_periph_ns[x]为低时，给外设请求接口 x 分配安全状态 (2) boot_periph_ns[x]为高时，给外设请求接口 x 分配非安全状态
boot_addr[31:0]	输入	硬连接线 32'h0	当 DMAC 从复位退出时，配置包含 DMAC 执行的第一条指令的地址的位置 注意，当 boot_from_pc 为高时，DMAC 只使用这个地址
boot_from_pc	输入	硬连接线 1'b0	当 DMAC 从复位退出时，控制 DMAC 执行它的初始化指令的位置 0：DMAC 等待来自一个 APB 接口的指令 1：DMA 管理器执行一个指令，该指令位于 boot_addr[31:0]所提供的地址的位置上

3.13.4 寄存器描述

表 3.41 给出了 DMAC 寄存器的概述。

表 3.41 DMA 控制器的寄存器

功　能	寄存器名字	概　述
DMAC 控制	dmac. XDMAPS_DS dmac. XDMAPS_DPC	提供安全状态和程序计数器
中断和事件	dmac. INT_EVENT_RIS dmac. INTCLR dmac. INTEN dmac. INTMIS	使能/禁止检测中断，屏蔽发送到中断控制器的中断，读原始的中断状态
故障状态和类型	dmac. FSRD dmac. FSRC dmac. FTRD dmac. FTR{7:0}	提供用于管理器和通道的故障状态和类型

续表

功　能	寄存器名字	概　述
通道线程状态	dmac. CPC{7:0} dmac. CSR{7:0} dmac. SAR{7:0} dmac. DAR{7:0} dmac. CCR{7:0} dmac. LC0_{7:0} dmac. LC1_{7:0}	这些寄存器提供了 DMA 通道线程的状态
调试	dmac. DBGSTATUS dmac. DBGCMD dmac. DBGINST{1,0}	这些寄存器使能用户发送指令到通道线程
IP 配置	dmac. XDMAPS_CR{4:0} dmac. XDMAPS_CRDN	这些寄存器使能系统固件发现 DMAC 的“硬连线”配置
看门狗	dmac. WD	当检测到一个锁定条件时，控制 DMAC 如何响应
系统级	slcr. DMAC_RST_CTRL slcr. TZ_DMAC_NS slcr. TZ_DMA_IRQ_NS slcr. TZ_DMAC_PERIPH_NS slcr. DMAC_RAM slcr. APER_CLK_CTRL	控制复位、时钟和安全状态

3.13.5　用于管理器和命令的指令集参考

表 3.42 和表 3.43 总结了 DMA 引擎指令和命令。

表 3.42　DMA 引擎指令总结

指　令	助　记　符	线程使用：M＝DMA 管理器	C＝DMA 通道
加半字	DMAADDH	—	C
加负半字	DMAADNH	—	C
结束	DMAEND	—	C
刷新和通知外设	DMAFLUSHP	—	C
去	DMAGO	M	—
杀死	DMAKILL	M	C
加载	DMALD	—	C
加载和通知外设	DMALDP	—	C
循环	DMAALP	—	C
循环结束	DMAALPEND	—	C
无限循环	DMAALPFE	—	C
移动	DMAMOV	—	C
无操作	DMANOP	M	C
读存储器屏障	DMARMB	—	C
发送事件	DMASEV	M	C
保存	DMAST	—	C
保存和通知外设	DMASTP	—	C
保存零	DMASEV	—	C

续表

指　令	助 记 符	线程使用：M=DMA 管理器　C=DMA 通道	
等待事件	DMAWFE	—	C
等待外设	DMAWFP	—	C
写存储器屏障	DMAWMB	—	C

表 3.43　汇编器提供的额外的命令

指　令	助 记 符
放置一个 32 位立即数	DCD
放置一个 8 位立即数	DCB
循环	DMALP
循环永远	DMALPFE
循环结束	DMALPEND
移动 CCR	DMAMOVCCR

3.13.6　编程模型参考

1. 存储器到存储器

这部分给出了 DMAC 执行的微码的例子，用于执行对齐、非对齐和固定传输。表 3.44 给出的是对齐传输；表 3.45 给出的是非对齐传输；表 3.46 给出的是固定传输。也给出了 MFIFO 的利用率。

表 3.44　对齐传输

描　述	代　码	MFIFO 使用
简单对齐程序：在这个程序中，源地址和目的地址对齐 AXI 数据总线宽度	DMAMOV CCR，SB4 SS64 DB4 DS64 DMAMOV SAR，0x1000 DMAMOV DAR，0x4000 DMALP 16 DMALD DMAST DMALPEND DMAEND	每个 DMALD 要求四个入口，每个 DMAST 移除四个入口；这个例子是一个静态的要求零 MFIFO 入口和动态的要求四个 MFIFO 入口
带有多个加载的对齐的非对称程序：下面的程序为每个保存执行四个加载，源地址和目的地址对齐 AXI 数据总线宽度	DMAMOV CCR，SB1 SS64 DB4 DS64 DMAMOV SAR，0x1000 DMAMOV DAR，0x4000 DMALP 16 DMALD DMALD DMALD DMALD DMAST DMALPEND	每个 DMALD 要求四个入口和每个 DMAST 移除一个入口；这个例子一个静态的要求零 MFIFO 入口和动态地要求四个 MFIFO 入口

续表

描　　述	代　　码	MFIFO 使用
带有多个保存的对齐的非对称程序：在这个程序中，源地址对齐 AXI 数据总线宽度，但是没有对齐目的地址。目的地址没有对齐到目标的猝发大小，这样第一个 DMAST 指令移出的数据比第一个 DMALD 指令所读取的数据少，因此要求一个单字的最终的 DMAST，用于清除来自 MFIFO 的数据	DMAMOV CCR，SB4 SS64 DB1 DS64 DMAMOV SAR，0x1000 DMAMOV DAR，0x4000 DMALP 16 DMALD DMAST DMAST DMAST DMAST DMALPEND DMAEND	每个 DMALD 要求四个入口，每个 DMAST 移除一个入口；这个例子是一个静态的要求零 MFIFO 入口和动态的要求四个 MFIFO 入口

表 3.45　非对齐传输

描　　述	代　　码	MFIFO 使用
对齐的源地址到非对齐的目的地址：在这个程序中，源地址对齐 AXI 数据总线宽度；但是目的地址没有对齐。目的地址没有对齐到目标的猝发大小，这样第一个 DMAST 指令移出的数据比第一个 DMALD 指令所读取的数据少，因此要求一个单字最终的 DMAST，用于清除来自 MFIFO 的数据	DMAMOV CCR，SB4 SS64 DB4 DS64 DMAMOV SAR，0x1000 DMAMOV DAR，0x4004 DMALP 16 DMALD DMAST DMALPEND DMAMOV CCR，SB4 SS64 DB1 DS32 DMAST DMAEND	第一个 DMALD 指令加载四个双字；但是由于目的地址非对齐，DMAC 将其移动四个字节，因此它使用 MFIFO 五个入口，每个 DMAST 只要求四个数据入口。因此，在程序期间，使用保留的额外入口，直到由最后的 DMAST 清空为止。这个例子是一个静态的要求一个 MFIFO 入口和动态的要求四个 MFIFO 入口
非对齐的源地址到对齐的目的地址：在这个程序中，源地址没有对齐 AXI 数据总线宽度。但是，对齐目的地址。源地址没有对齐到源猝发大小，这样第一个 DMALD 指令读出的数据比 DMAST 所要求的要少。因此，要求一个额外的 DMALD 来满足第一个 DMAST	DMAMOV CCR，SB4 SS64 DB4 DS64 DMAMOV SAR，0x1004 DMAMOV DAR，0x4000 DMALD DMALP 15 DMALD DMAST DMALPEND DMAMOV CCR，SB1 SS32 DB4 DS64 DMALD DMAST DMAEND	第一个 DMALD 指令加载 5 拍数据，使能 DMAC 执行第一个 DMAST。当第一个 DMALD 后，随后的 DMALD 并没有对齐源猝发大小。例如，第二个 DMALD 从地址 0x10 读。当循环后，最后两个 DMALD 读取要求的数据，以满足最后的 DMAST。这个例子是一个静态的要求一个 MFIFO 入口和动态的要求四个 MFIFO 入口

续表

描　　述	代　　码	MFIFO使用
非对齐的源地址到对齐的目的地址，超过最初的加载：这个程序是对前面所描述的非对齐的源地址和对齐的目的地址的一个替代。程序使用了一个不同的源猝发序列，可能不是高效的，但是要求较少的MFIFO入口	DMAMOV CCR，SB5 SS64 DB4 DS64 DMAMOV SAR，0x1004 DMAMOV DAR，0x4000 DMALD DMAST DMAMOV CCR，SB4 SS64 DB4 DS64 DMALP 14 DMALD DMAST DMALPEND DMAMOV CCR，SB3 SS64 DB4 DS64 DMALD DMAMOV CCR，SB1 SS32 DB4 DS64 DMALD DMAST DMAEND	第一个DMALD指令加载5拍数据，使能DMAC执行第一个DMAST。当第一个DMALD后，随后的DMALD并没有对齐源猝发大小，例如，第二个DMALD从地址0x10读。当循环后，最后两个DMALD读取要求的数据满足最后的DMAST。这个例子是一个静态的要求一个MFIFO入口和动态的要求四个MFIFO入口
对齐的猝发大小，非对齐的MFIFO：在这个程序中，目的地址比MFIFO的宽度要窄，对齐猝发大小。但是，没有对齐MFIFO宽度	DMAMOV CCR，SB4 SS32 DB4 DS32 DMAMOV SAR，0x1000 DMAMOV DAR，0x4004 DMALP 16 DMALD DMAST DMALPEND DMAEND	如果DMAC配置有一个32位的AXI数据总线宽度，则这个程序要求四个MFIFO入口。然而，在这个例子中，DMAC有一个64位的AXI数据宽度，因为目的地址不是64位对齐，它要求3个而非两个MFIFO入口。这个例子是一个静态的要求零个MFIFO入口和动态的要求三个MFIFO入口

表3.46　固定传输

描　　述	代　　码	MFIFO使用
固定目的对齐地址：在这个程序中，源地址和目的地址对齐AXI数据总线宽度，目的地址是固定的	DMAMOV CCR，SB2 SS64 DB4 DS32 DAF DMAMOV SAR，0x1000 DMAMOV DAR，0x4000 DMALP 16 DMALD DMAST DMALPEND DMAEND	程序中的每个DMALD加载两个64位的数据传输到MFIFO。由于目的地址是32位的固定地址，则DMAC将每个64位的数据项分割穿过MFIFO的两个入口。这个例子是一个静态的要求零个MFIFO入口和动态的要求四个MFIFO入口

2. 存储器和外设之间

这里有两个不同的方法处理 DMAC 和外设之间的数据流的数量。

1）外设长度管理

外设控制一个 DMA 周期内所包含的数据的数量，DMAC 无须知道它包含所需要传输数据的个数。

2）DMAC 长度管理

DMAC 控制一个 DMA 周期内数据的数量。

下面分别给出外设长度管理和 DAMC 长度管理的例子。

1）外设长度管理

下面的例子给出了一个 DMAC 程序，当外设发送一个猝发请求（DMA{3:0}_DRTYPE ＝b01）时，从存储器传输 64 个字到外设 0。当外设发送单个请求（DMA{3:0}_DRTYPE ＝b00）时，DMAC 程序从存储器传输 1 个字到外设零。

为了传输 64 个字，程序引导 DAMC 执行 16 个 AXI 传输。每个 AXI 传输由 4 拍猝发组成（SB＝4，DB＝4），每一拍移动一个数据字（SS＝32，DS＝32）。

在这个例子中，程序给出了下面指令的用法：

（1）DMAWFP 指令。DMAC 等待来自外设的猝发或者单个请求；

（2）DMASTPB 和 DMASTPS 指令。当传输完成后 DMAC 通知外设。

代码清单 3-1

```
# 设置猝发传输(4 拍猝发,于是 SB4 和 DB4),
# (字数据宽度, 于是 SS32 和 DS32)
DMAMOV CCR SB4 SS32 DB4 DS32
DMAMOV SAR ...
DMAMOV DAR ...
# 初始化外设 '0'
DMAFLUSHP P0
# 执行外设传输
# 外部循环 - DMAC 响应外设请求,直到外设设置 drlast_0 = 1
DMALPFE
# 等待请求,DMAC 设置 request_type0 标志,取决于它接收到的请求类型
DMAWFP 0, periph
# 为猝发请求设置循环: 16 个交易最开始的 15 个
# 注意: B 后缀-有条件的执行,只有如果 request_type0 标志为 burst
DMALP 15
DMALDB
DMASTB
# 如果服务一个猝发,循环返回,否则当作一个 NOP
DMALPENDB
# 执行最后一个交易(16 个中的第 16 个),给外设发送猝发请求完成响应
DMALDB
DMASTPB P0
#如果外设发送单个请求信号,执行交易
# 注意: S 后缀-有条件的执行,只有如果 request_type0 标志为 Single
DMALDS
DMASTPS P0
# 如果 DMAC 接收到最后请求,即 drlast_0 = 1,退出循环
DMALPEND
DMAEND
```

2）DMAC长度管理

这个例子给出了当外设发送16个连续的猝发请求和3个连续的单个请求时，一个DMAC程序传输1027个字。

代码清单3-2

```
# 设置AXI猝发传输
# (4拍猝发,于是SB4和DB4),(字数据宽度,于是SS32和DS32)
DMAMOV CCR SB4 SS32 DB4 DS32
DMAMOV SAR ...
DMAMOV DAR ...
# 初始化外设'0'
DMAFLUSHP P0
#执行外设传输
# 猝发请求循环,传输1024个字
DMALP 16
# 等待外设发送一个猝发请求信号
# DMAC传输64个字,用于每个猝发请求
DMAWFP 0, burst
# 为猝发请求设置循环：16个交易的最开始的15个
DMALP 15
DMALD
DMAST
DMALPEND
# 执行最后的交易(16个交易中的第16个)
#发送猝发请求完成的外设相应信号
DMALD
DMASTPB 0
#完成猝发循环
DMALPEND
# 设置AXI单个传输(字数据宽度，于是SS32和DS32)
DMAMOV CCR SB1 SS32 DB1 DS32
#单个请求循环传输3个字
DMALP 3
# 等待外设发送一个单个请求信号. DMAC传输一个字
DMAWFP 0, single
#为单个请求执行交易,并且发送完成响应信号到外设
DMALDS
DMASTPS P0
#完成单个循环
DMALPEND
# 刷新外设,防止单个传输响应一个猝发请求
DMAFLUSHP 0
DMAEND
```

3. 使用一个事件重新启动DMA通道

当编程INTEN寄存器产生一个事件时，DMASEV和DMAWFE指令可以用来重新启动一个或者多个DMA通道。

下面给出了出现这些情况时，DMAC的行为：

（1）在DMASEV之前，DMAC执行DMAWFE；

（2）在DMAWFE之前，DMAC执行DMASEV。

1）在 DMASEV 之前，DMAC 执行 DMAWFE

重新启动一个单个 DMA 通道：

（1）第一个 DMA 通道执行 DMAWFE 指令。然后停止，等待发生事件。

（2）其他通道使用相同的事件号，执行 DMASEV 指令。这样产生一个事件，重新启动第一个 DMA 通道。它执行完一个 DMASEV 后，DMAC 需要一个 DMA{3:0}_ACLK 周期清除事件。

可以编程多个通道，用来等待相同的事件。例如，如果四个 DMA 通道为事件 12，都执行 DMAWFE，则当另一个 DMA 通道为事件 12 执行 DMASEV 时，在同一时间重新启动所有的四个 DMA 通道。它执行完一个 DMASEV 后，DMAC 需要一个时钟周期清除事件。

2）在 DMAWFE 之前，DMAC 执行 DMASEV

在另一个通道执行 DMAWFE 之前，如果 DMAC 执行了 DMASEV，则事件保持等待处理，直到 DMAC 执行 DMAWFE。当 DMAC 执行 DMAWFE 时，它停止执行一个 DMA{3:0}_ACLK 周期，清除事件，然后继续执行通道线程。

例如，如果 DMAC 执行 DMASEV6，并且没有其他线程执行 DMAWFE6，则事件保持等待处理。如果 DMAC 为通道 4 执行了 DMAWFE6 指令，然后为通道 3 执行了 DMAWFE6 指令，则

（1）DMAC 停止执行通道 4 线程，一个 DMA{3:0}_ACLK 周期；

（2）DMAC 清除事件 6；

（3）DMAC 继续执行通道 4 的线程；

（4）当执行 DMASEV 后，DMAC 停止通道周期的执行。

4. 中断一个处理器

DMAC 提供了 irq[7:0]信号，用于到外部微处理器的高电平敏感的中断。当编程 INTEN 寄存器用于产生一个中断时，在 DMAC 执行 DMASEV 后，它将相应的 irq[7:0]信号设置为高。

通过写中断清除寄存器，一个外部处理器可以清除中断。

执行 DMAWFE 不能清除中断。

当 DMAC 完成一个 DMALD 或者 DMAST 指令时，如果 DMASEV 指令用于通知一个微处理器，ARM 推荐在 DMASEV 之前，插入一个存储器屏障指令。否则，DMAC 可能在 AXI 传输完成以前发送一个中断信号。

下面的例子说明了这个问题。

代码清单 3-3

```
DMALD
DMAST
# 发布一个写存储器屏障
#在 DMAC 能发布一个中断前，等待完成 AXI 写传输
DMAWMB
# DMAC 发布中断
  DMASEV
```

3.13.7 编程限制

下面给出了两个编程限制。

1. 在一个 DMA 周期内，更新 DMA 通道控制寄存器

在 DMAC 执行 DMALD 和 DMAST 序列前，用户写入 CCRn 寄存器、SARn 寄存器和 DARn 寄存器的值，用来在 DMAC 执行的过程中（即将数据从源地址传输到目的地址），控制对数据字节通道的操作。

可以在一个 DMA 周期内更新这些寄存器。但是，如果修改了某个寄存器的域，DMAC 可能放弃数据。下面描述了寄存器域可能对数据传输有一个不利的影响。

1）更新影响目的地址

如果使用 DMAMOV 指令，更新 DARn 或者 CCRn 寄存器，一部分是在 DMA 周期，可能会发生目标数据流的不连续。如果下面的任何一个发生变化，则发生不连续：

（1）dst_inc 比特；

（2）当 dst_inc=0（固定地址猝发），dst_burst_szie 域；

（3）DARn 寄存器，它修改目的字节通道对齐，例如，当总线宽度为 64 位，修改 DARn 寄存器的[2:0]位。

当发生目的数据流的不连续时，DMAC：

（1）停止执行 DMAC 通道线程；

（2）完成为通道的所有读和写操作（只是看上去好像 DMAC 执行 DMARMB 和 DMAWMB 指令）；

（3）放弃任何驻留在 MFIFO 内的数据；

（4）继续执行 DMA 通道线程。

2）更新影响源地址

如果使用 DMAMOV 指令，更新 SARn 或者 CCRn 寄存器，一部分是在 DMA 周期，可能会发生源数据流的不连续。如果下面的任何一个发生变化，则发生不连续：

（1）src_inc 比特；

（2）src_burst_szie 域；

（3）SARn 寄存器，它修改目的字节通道对齐，例如，当总线宽度为 32 位，修改 SARn 寄存器的[1:0]位。

当发生目的数据流的不连续时，DMAC：

（1）停止执行 DMAC 通道线程；

（2）完成为通道的所有读操作（只是看上去好像 DMAC 执行 DMARMB 指令）；

（3）继续执行 DMA 通道线程。没有丢弃来自 MIFIFO 的数据。

2. 在 DMA 通道内，共享资源

DMA 通道程序共享 MFIFO 数据存储资源。一个并发运行的 DMA 通道程序集，在启动时，不能要求资源超过 MFIFO 的大小。如果超过了这个限制，DMAC 可能锁定和产生一个看门狗退出。

DMAC 包含一个称为加载-锁机制，确保正确地使用 MFIFO 资源。加载-锁或者由一个通道拥有，或者它是自由的。拥有加载-锁的通道能成功地执行 DMALD 指令。没有拥有加载-锁的通道，在执行 DMALD 指令时暂停，直到它拥有加载-锁。

当下面情况时，一个通道声明拥有加载锁：

(1) 它执行一个 DMALD 或者 DMALDP 指令；

(2) 当前没有通道拥有加载-锁。

当下面情况时，一个通道释放对加载-锁的拥有：

(1) 它执行 DMAST、DMASTP 或者 DMASTZ；

(2) 它达到一个障碍，即它执行 DMARMB 或者 DMAWMB；

(3) 它等待，即它执行 DMAWFP 或者 DMAWFE；

(4) 它正常地终止，即它执行 DMAEND；

(5) 由于任何原因的退出，包括 DMAKILL。

一个 DMA 通道程序对 MFIFO 资源的使用是在 MFIFO 入口测量。当处理程序的时候，增加或者减少。通过使用静态要求和动态要求(该要求被加载-锁影响)，描述一个 DMA 通道程序所要求的 MFIFO 资源。

ARM 定义在通道做下面事情前，静态要求一个通道当前正在使用 MIFIFO 入口的最大数目：

(1) 执行一个 WFP 或者 WFE 指令；

(2) 声称拥有加载-锁。

ARM 定义了动态要求，即任何时候在执行一个通道程序时，要求的最大 MFIFO 入口的数量。

计算总的 MFIFO 要求，将最大的动态要求增加到所有静态要求的和。

为了避免 DMAC 锁定，通道程序所要求总的 MFIFO 大小必须等于或者小于最大的 MFIFO 深度。DMAC 最大的 MFIFO 深度为每个一个字(64 位)。

3.13.8 DMAC IP 配置选项

Xilinx 使用表 3.47 的 IP 配置选项实现 DMAC。

表 3.47 DMAC IP 配置选项

IP 配置选项	值	IP 配置选项	值
数据宽度(比特)	64	读队列深度	16
通道个数	8	写队列深度	16
中断个数	16(8 中断，8 事件)	读发布能力	8
外设个数	4(到可编程逻辑)	写发布能力	8
缓存行个数	8	外设请求能力	所有能力
缓存行宽度(字)	4	安全 APB 基地址	0xF800_3000
缓冲深度(MFIFO 深度)	1	非安全 APB 基地址	0xF800_4000

思考题 3-18：请说明 Zynq-7000 内的 DMA 控制器的结构特点及实现什么功能？