**ARMv8处理器学习报告**

贾文超 2253968 2253968@tongji.edu.cn

卞思涵 2253892 2253892@tongji.edu.cn

张扬 2253722 2253722@tongji.edu.cn

苗君文 2253893 2253893@tongji.edu.cn

**摘要**：本报告详细探讨了ARMv8架构处理器的关键设计和实现，涵盖了浮点运算单元（FPU）和NEON技术、内存管理单元（MMU）、流水线结构和缓存设计。通过对AArch64 NEON架构的改进，ARMv8处理器在媒体编解码、图形处理及科学计算等应用中展现出卓越的性能。同时，内存管理单元通过有效的虚拟地址和物理地址转换，实现了多任务处理的支持。ARMv8处理器的流水线结构和缓存设计，通过优化数据处理流程，提高了系统的整体性能和能效比。

# 一、 引 言

ARMv8架构作为ARM处理器家族的成员，继承了ARM架构一贯的高效和低功耗设计理念。随着科技的进步和计算需求的增加，ARMv8在性能和能效方面进行了显著提升。尤其是在浮点运算单元（FPU）和NEON技术的改进上，使其在多媒体处理和科学计算等领域表现突出。此外，ARMv8的内存管理单元（MMU）通过高效的虚拟地址到物理地址转换和灵活的缓存策略，进一步提高了系统的稳定性和响应速度。本报告将通过对ARMv8架构的浮点运算单元、内存管理单元、流水线结构和缓存设计的深入分析，展示其在现代计算领域中的优势和应用前景。

# 二、 浮点运算单元（FPU）设计

ARM高级SIMD（单指令多数据）架构及其相关实现和支持软件，通常被称为NEON技术。NEON指令集适用于AArch32（对应ARMv7的NEON指令）和AArch64。通过NEON技术，ARM架构能够显著加速在大数据集上执行的重复操作，在媒体编解码、图形处理以及科学计算等应用中有重要作用。

AArch64的NEON架构使用32个128位寄存器，是ARMv7中寄存器数的两倍。这些寄存器也被浮点指令使用。所有编译的代码和子程序都遵循嵌入式应用程序二进制接口（EABI）的规范，EABI规定了哪些寄存器可以在子程序执行过程中被修改，以及哪些寄存器必须在特定子程序执行期间保持其原始值。编译器可以在代码的任何位置自由使用任何NEON/VFP寄存器来存储浮点值或NEON数据。

尽管在所有的标准ARMv8实现中，浮点运算和NEON技术都是必需的组成部分，但针对特定市场的实现可能提供不同的功能组合：

* 不支持NEON或浮点运算的简化实现。
* 支持全浮点和SIMD运算，并具备异常捕获能力的实现。
* 支持全浮点和SIMD运算，但不包含异常捕获功能的实现。

**2.1 AArch64 中的 NEON 和浮点运算的新特性**

在AArch64架构中，NEON 相对于AArch32 NEON，进行了以下更改：

* **寄存器扩展**：现在有 32 个 128 位寄存器，而 ARMv7 只有 16 个。
* 寄存器映射优化：较小的寄存器不再被打包到较大的寄存器中，而是以一对一的方式映射到 128 位寄存器的低位。单精度浮点值使用低 32 位，而双精度浮点值使用 128 位寄存器的低 64 位。
* **指令前缀变更**：ARMv7-A NEON 指令中的 V 前缀已被移除。
* **写入操作**：对矢量寄存器的 64 位或更小的写入操作会将高位清零。
* **新增指令**：包括通道插入和提取指令，以及用于操作 128 位矢量寄存器高 64 位的指令，提供了更丰富的数据处理功能。
* **矢量归约操作**：新增了跨通道求和、最小值和最大值的归约操作，有助于复杂数据处理。
* **指令扩展**：部分现有指令已扩展以支持 64 位整数值，包括比较、加法、绝对值和取反等操作。
* **饱和指令扩展**：饱和指令现支持无符号累加到有符号和有符号累加到无符号的操作。
* **双精度浮点支持**：AArch64 NEON 支持双精度浮点和完整的 IEEE754 操作，包括舍入模式、非规范化数和 NaN 处理。

AArch64 中的浮点运算进行了以下增强：

* **指令前缀更新**：ARMv7-A 浮点指令中的 V 前缀已被 F 前缀取代。
* IEEE754标准支持：支持 IEEE 754 浮点标准定义的单精度（32 位）和双精度（64 位）浮点矢量数据类型和算术操作，遵守 FPCR 舍入模式字段、默认 NaN 控制、置零控制，以及（如果实现支持）异常捕获使能位。
* **寻址模式统一**：FP/NEON 寄存器的加载/存储寻址模式与整数加载/存储相同，包括加载或存储一对浮点寄存器的能力。
* **条件选择指令**：新增了浮点 FCSEL 和选择与比较指令，提供了更灵活的条件分支控制。
* **舍入模式支持**：浮点乘加和乘减指令均为融合操作，支持多种定向舍入模式，减少了精度损失。
* **转换操作扩展**：增加了多种数据类型之间的转换操作，如 64 位整数与浮点、半精度与双精度之间的转换。
* **浮点舍入指令**：新增了浮点舍入到最接近整数的指令（FRINTx），支持多种舍入模式。
* **下转换指令优化**：双精度到单精度下转换指令（FCVTXN）支持不精确舍入到奇数，适合继续下转换到半精度。
* **数值比较指令**：新增了 FMINNM 和 FMAXNM 指令，支持 IEEE 754-2008 标准的 minNum() 和 maxNum() 操作。
* **浮点矢量归一化加速**：新增指令 FRECPX 和 FMULX 用于加速浮点矢量归一化操作。

**2.2 NEON和浮点架构**

NEON 寄存器是用于存储向量数据的特殊寄存器，其中每个向量包含相同数据类型的一组元素。向量被分割成多个通道，每个通道包含一个数据元素。

NEON 向量的通道数量取决于向量的大小和向量中的数据元素。通常每条 NEON 指令会导致 n 个操作并行执行，其中 n 是输入向量被分成的通道数量。一个通道的进位或溢出不会影响其他通道。向量中元素的排列顺序是从最低有效位开始的，即元素 0 使用寄存器的最低有效位。

NEON 和浮点指令操作的元素类型包括：

* 32 位单精度和 64 位双精度浮点数。
* 16 位浮点数：仅作为转换的格式。它不支持用于数据处理操作。
* 8 位、16 位、32 位或 64 位无符号和有符号整数。
* 8 位和 16 位多项式。

多项式类型用于使用二次幂有限域或 {0,1} 上的简单多项式的代码，例如纠错。正常的 ARM 整数代码通常使用查找表进行有限域算术。AArch64 NEON 提供了使用大查找表的指令。

NEON 单元将寄存器文件视为2种情况：

* 32 个 128 位四字寄存器，V0-V31：

图表, 散点图

描述已自动生成

图2.1 32 个 128 位四字寄存器

* 32个 64 位双字寄存器，D0-D31：

表格

中度可信度描述已自动生成

图2.2 32个 64 位双字寄存器

在任何时候，这些寄存器都是完全可访问的。软件不需要显式地在四字和双字寄存器之间进行切换，因为所使用的指令会自动选择适当的寄存器视图。这种灵活性使得 NEON 架构能够高效地处理各种数据大小和类型。

**2.3 浮点单元**

在 AArch64 架构中，浮点单元（FPU）与 NEON SIMD（单指令多数据）指令集紧密集成，提供了对浮点数据的广泛支持。在 FPU 和 NEON 的上下文中，寄存器文件被划分为多个不同大小和精度的视图，以支持多种数据类型和操作。

* D 寄存器（双精度）：包含 32 个 64 位宽的寄存器，编号为 D0 到 D31。这些寄存器被称为双精度寄存器，用于存储双精度浮点值。
* S 寄存器（单精度）：包含 32 个 32 位宽的寄存器，编号为 S0 到 S31。这些寄存器被称为单精度寄存器，用于存储单精度浮点值。
* H 寄存器（半精度）：包含 32 个 16 位宽的寄存器，编号为 H0 到 H31。这些寄存器被称为半精度寄存器，尽管在实际硬件中可能并不直接支持半精度浮点值的原生存储，但它们提供了一种逻辑上的视图，以支持对 16 位数据的操作。
* 上述寄存器的组合。

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

图2.3 浮点寄存器划分

**2.4 标量数据和NEON**

标量数据在 NEON 指令集中指的是单独的数据值，而非构成向量的多个数据值的集合。一些 NEON 指令使用标量操作数。寄存器内的标量通过索引访问向量值。

访问 NEON 寄存器中单个元素的通用表示法如下：

<Instruction> Vd.Ts[index1], Vn.Ts[index2]

其中：

* Vd：目标寄存器，用于存储操作结果。
* Vn：源寄存器，提供操作所需的数据。
* Ts：元素的大小说明符。
* Index：元素索引。

例如，在下面的指令中：

INS V0.S[1], V1.S[0]

这条指令将寄存器 V1 中单精度浮点数视图（S）的第一个元素（索引为 0）复制到寄存器 V0 的单精度浮点数视图的第二个元素（索引为 1）中。

图表

描述已自动生成

图2.4 INS V0.S[1], V1.S[0]指令示例

在MOV V0.B[3], W0指令中，将通用寄存器 W0 的最低有效字节复制到 NEON 寄存器 V0 的字节视图（B）的第四个元素（索引为 3）中。

图示

低可信度描述已自动生成

图2.5 MOV V0.B[3], W0指令示例

NEON 标量可以是 8 位、16 位、32 位或 64 位值。除了乘法指令外，访问标量的指令可以访问寄存器文件中的任何元素。乘法指令仅允许 16 位或 32 位标量，并且仅能访问寄存器文件中的前 128 个标量：

* 16 位标量仅限于寄存器 Vn.H[x]，其中 0 ≤ n ≤ 15。
* 32 位标量仅限于寄存器 Vn.S[x]。

**2.5浮点参数**

浮点值通过浮点寄存器传递给函数，也可以通过浮点寄存器被函数返回。整数（通用）寄存器和浮点寄存器可以同时使用。这意味着浮点参数通过浮点 H、S 或 D 寄存器传递，而其他参数则通过整数 X 或 W 寄存器传递。AArch64 过程调用标准要求在需要浮点运算的地方使用硬件浮点，因此在 AArch64 状态中不存在软件浮点链接。

主要的浮点数据处理操作如下所示：

文本

低可信度描述已自动生成

表格

描述已自动生成

图2.6 主要浮点数据处理操作

# 三、 内存管理单元

**3.1 总览**

内存管理单元的重要功能：使系统能够同时运行多个任务。独立的程序在自己的私有虚拟内存空间中运行，不需要了解系统的物理内存映射，也不需要了解其他程序的地址。

单个系统中不同处理器和设备会有**不同的虚拟地址和物理地址映射**，操作系统通过对MMU进行编程以在这两种视图之间进行切换。

MMU将代码和数据的虚拟地址转换为实际系统中的物理地址。翻译是在硬件上自动进行的，并且对应用程序是透明的。除了地址转换之外，MMU还控制每个内存区域的内存访问权限、内存序和缓存策略。

MMU允许在编写任务或应用程序时，**不需要了解系统的物理内存映射或可能同时运行的其他程序**。这允许你为每个程序使用相同的虚拟内存地址空间。

它还允许使用连续的虚拟内存映射，即使物理内存是碎片状的。这个虚拟地址空间与系统中内存的实际物理映射是分开的。应用程序被编写、编译和链接以在虚拟内存空间中运行。

**3.2 转换后备缓冲区（TLB）**

转换后备缓冲区(TLB)是MMU中一个**最近访问的页面转换关系的缓存**。对于处理器执行的每次内存访问，MMU检查其转换关系是否缓存在TLB中。如果请求的地址转换在TLB内命中，则地址转换立即可用。

每个TLB条目通常不仅包含物理地址和虚拟地址，还包含内存类型、缓存策略、访问权限、地址空间ID (ASID)和虚拟机 ID (VMID)。如果由处理器发出的地址在TLB中没有有效的转换，称为TLB缺失，那就要执行外部转换表遍历或查找。MMU中的专用硬件使其能够读取内存中的转换表。然后，如果转换表遍历没有导致页面错误，那么可以将新加载的转换缓存在TLB中，以便可能的重用。TLB的确切结构在不同的ARM处理器实现中是不同的。

ARMv8-A体系结构提供了一个称为**连续块条目**（contiguous block entries ）的特性来有效地使用TLB空间。转换表块的每个条目都包含一个连续的位。设置后，这个位向TLB发信号，它可以缓存单个条目来覆盖多个块转换关系。查找可以索引到连续块所覆盖的地址范围的任何位置。因此TLB可以为一个定义好的地址区间只缓存一个条目，这样就可以在TLB中存储比其他情况下更大范围的虚拟地址。

要使用一个连续位，连续的块必须是相邻的，即**它们必须对应于一个连续的虚拟地址范围**。它们必须从对齐的边界开始，具有一致的属性，并指向同一级转换的连续输出地址范围。

**更加详细的理解如下：**

* 块条目：这是指一个地址转换条目，它可以覆盖一块连续的内存区域，而不是一个单独的内存页。
* 连续位：在块条目中有一个特殊的“连续位”。如果这个位被设置，意味着这个条目不只表示一个单独的块，而是多个连续的块。
* TLB缓存优化：当连续位被设置后，TLB会知道这个条目不仅适用于一个地址块，而是适用于多个连续的块。因此，TLB可以用一个条目来表示多个块的地址转换关系。
* 范围覆盖：当系统查找一个虚拟地址时，如果这个地址落在连续块的范围内，TLB只需要一个条目就可以找到对应的物理地址。这意味着，在相同的TLB大小下，连续块条目可以覆盖更大的虚拟地址范围。

这样一来，连续块条目就能帮助减少TLB条目的数量，从而提高TLB的效率和内存访问速度。简而言之，它允许TLB用一个条目来表示多个连续的地址块，从而更高效地利用TLB空间。

**3.3 虚拟地址到物理地址的转换**

由于虚拟地址是64位，而实际物理地址统一使用48位寻址，因此在实际转换过程中，对于一个虚拟地址的高16位[63:47]必须全是0或者1，否则会发生错误。

不过在ARMv8中，虚拟地址的高8位可以被忽略，从而使这些位用作其他的一些用途。

一般来说，虚拟地址的高16位如果全是0或者全是1分别会代表着使用TTBR0或者TTBR1（两个不同的转换表基寄存器，意味着使用不同的转换表），虚拟地址的后几位便使用多级页表的转换方式。

图示

描述已自动生成

图3.1 虚拟地址到物理地址的转换示例

值得注意的是，每个二级表可以与一个或多个一级表关联，这意味着可以将多个虚拟位置作为相同物理地址的别名。

**3.4 ARMv8-A的转换表**

表的描述符格式如下，其类型用最低的两位0：1来标识：

图形用户界面, 表格

中度可信度描述已自动生成

图3.2 转换表的描述符格式

其中，0级表只能输出1级表的地址，第3级只能输出块地址。

**颗粒（页）大小对转换表的影响**

1. 颗粒大小4KB时，硬件使用4级查找

文本, 表格

描述已自动生成

图3.3 颗粒大小4KB时的转换表

1. 颗粒大小为16KB时，硬件使用4级查找，每一级转换的地址位数量与4KB时有所不同

文本, 表格

描述已自动生成

图3.4 颗粒大小为16KB时的转换表

1. 颗粒大小为64KB时，硬件使用3级查找

文本, 表格

描述已自动生成

图3.5 颗粒大小为64KB时的转换表

# 四、 ARMv8流水线结构

**4.1 ARM流水线的发展历程**

ARM的流水线结构经历了3级流水线、5级流水线、6级流水线到8级流水线的发展。从ARM7TDMI处理器到使用ARMv6架构的ARM11及更高级结构的其他处理器，处理器性能随着流水线段数的增加和结构的改进显著提升。值得一提的是，ARM架构的流水线设计简洁而不失效率，主要采用静态按序调度，并没有使用动态执行、乱序执行、推测式执行等过多复杂的优化。本文以基于ARMv8架构的Cortex-A53处理器为例，简述ARMv8架构的流水线结构。

图示, 工程绘图

描述已自动生成

图4.1 ARM流水线的发展

ARM Cortex-A53是一款基于ARMv8指令系统的八级流水线结构处理器，核功耗不超过0.13W，主频可达1.5Ghz，曾是世界上能耗比最高、面积最小的64位应用处理器。该处理器采用了动态多发射技术，即每一周期可发射两条指令。同时，其属于静态按序流水线。Cortex-A53处理器的流水线结构主要分为取指和预测、指令译码、定点执行和访存操作、浮点执行四个部分。

图示

描述已自动生成

图4.2 Cortex-A53处理器的流水线结构

**4.2 取指和预测**

该部分负责从指令缓存中取出指令，并进行指令预测。该部分包含的部件有AGU+TLB，指令缓存（Instruction Cache），混合预测器（Hybrid Predictor）和间接预测器（Indirect Predictor）。

其中，AGU（Address Generation Unit）为地址生成单元，负责计算当前需要获取的指令地址。这通常包括PC（程序计数器）的值以及任何需要加到PC上的偏移量。TLB（Translation Lookaside Buffer）为翻译后备缓冲区，其将虚拟地址转换为物理地址，以便在缓存中快速找到指令。指令缓存（Instruction Cache）存储最近使用的指令，以减少从主存读取指令的延迟。该部件通过TLB转换得到的物理地址在指令缓存中查找，如果命中，则读取指令。混合预测器（Hybrid Predictor）结合了多种分支预测技术，以提高分支预测的准确性。它可以根据历史执行情况预测下一条指令的位置。AGU和TLB提供的地址用于指令缓存查找，同时预测器根据当前指令和历史信息预测下一条指令的位置，它们之间的操作是并行的。间接预测器（Indirect Predictor）专门用于预测间接跳转的目标地址。例如，函数指针或虚函数调用的目标地址。这种预测器通过记录和分析历史间接跳转的目标地址，提高预测的准确性。

该部分的工作流程可分为F1、F2、F3、F4共四个阶段。F1阶段的任务是生成指令地址，AGU生成当前PC指向的指令地址，TLB将虚拟地址转换为物理地址。F2阶段的任务是查找和获取指令，指令缓存使用物理地址查找对应的指令。若命中，则读取指令；若未命中，则从主存加载指令。F3阶段的任务是进行指令预测，混合预测器对当前指令进行分支预测，间接预测器预测间接跳转的目标地址。F4阶段的任务是准备下一周期的指令获取，更新程序计数器PC，指向下一条要获取的指令地址。获取的指令被放入指令队列，等待译码阶段处理。

**4.3 指令译码**

该部分负责对接收上一部分传递的指令并进行译码。工作流程可分为D1、D2、D3共三个阶段。D1阶段进行初步译码，负责识别指令的大致类型，例如算术运算，访存指令，逻辑运算等。初步译码后的指令将被放入13条指令队列中，暂存以供进一步处理。指令队列的存在有助于平衡取指阶段和详细译码阶段之间的负载，避免因取指阶段的速度波动导致的停顿。D2阶段进行主要译码，负责从指令队列中取出指令，进行详细译码。详细译码包括确定操作数、目标寄存器和具体的操作类型，并将译码结果传递到后期译码阶段。D3阶段进行后期译码，负责处理复杂指令的特殊需求，确保所有指令在进入执行阶段前都准备就绪。

**4.4 定点执行和访存操作**

该部分负责定点执行和访存操作。其包含了许多相关部件，如发射单元，整数寄存器文件，各类执行流水线等。其中，发射单元（Issue）负责将译码后的指令分发到不同的执行单元。发射单元会根据指令类型和操作数的可用性决定指令的执行顺序。整数寄存器文件（Integer Register File）是存储整数操作数的寄存器文件。所有整数运算都会从这里读取操作数并将结果写回这里。各类执行流水线则分工不同的职能。ALU pipe 0和 ALU pipe 1是两个算术逻辑单元（ALU）流水线，执行简单的整数运算，如加法、减法、逻辑运算等。乘加单元流水线（MAC pipe）执行乘法和乘加运算。除法单元流水线（Divide pipe）执行除法运算。加载单元流水线（Load pipe）负责从内存读取数据。存储单元流水线（Store pipe）负责将数据写入内存。

该部分的工作流程可分为Iss，Ex1，Ex2，Wr四个阶段。Iss阶段的任务是指令分发，发射单元根据指令类型和操作数的可用性将指令分发到不同的执行流水线。Ex1阶段开始执行指令，各执行单元进行初步运算或内存地址计算。Ex2阶段完成指令执行，各执行单元完成运算，将结果存储在临时寄存器或更新内存地址。Wr阶段写回执行结果，将运算结果写回到相应的寄存器或内存，完成指令执行过程。

**4.5 浮点执行**

该部分进行浮点运算操作，其具有NEON寄存器文件（NEON Register File），浮点算术逻辑单元流水线（ALU pipe）等关键部件。其中，NEON寄存器文件（NEON Register File）是存储浮点和高级SIMD（单指令多数据）操作数的寄存器文件。所有浮点和SIMD运算都会从这里读取操作数并将结果写回这里。浮点和SIMD执行流水线由两类流水线组成，一类为浮点乘法、除法和平方根运算单元流水线（MUL/DIV/SORT pipe），另一类为浮点算术逻辑单元流水线（ALU pipe）, 执行浮点加法、减法和其他基本浮点运算。

该部分的工作流程可分为F1,F2,F3,F4,F5共五个阶段。这里的“F”意为Floating Point Execution，与取值和预测部分的“F”（Fetch）含义不同。其中，F1阶段的任务是准备浮点运算，从NEON寄存器文件中读取浮点或SIMD指令的操作数。F2阶段开始执行浮点指令，即开始执行浮点乘法、除法、平方根运算或基本浮点运算。F3阶段进行中间计算，继续进行浮点运算的中间步骤。F4阶段完成浮点运算，将结果存储在临时寄存器中。F5阶段回写浮点运算结果，将浮点和SIMD运算的结果写回到NEON寄存器文件，完成指令执行过程。

Cortex-A53处理器采用静态按序的八级流水线结构，分为多个阶段来处理指令，实现了高性能和低功耗的平衡。其流水线结构堪称顺序流水线的卓越之作，体现了ARMv8架构的优秀设计。

# ARMv8缓存设计

**5.1 ARMv8缓存层次结构**

典型的CPU缓存结构是由一级缓存(L1 Cache)和二级缓存(L2Cache)组成，部分高端CPU还具有三级缓存(L3Cache)[1]。ARMv8-A架构处理器常采用多级缓存系统，每个核心通常包括较小的L1指令和数据缓存，而Cortex-A53和Cortex-A57处理器则常具备两级或更多级缓存结构，包括较小的L1缓存和一个更大的共享L2缓存，使得多核簇中的核心可以共享L2缓存，同时可能存在外部L3缓存用于簇间共享。

图示

描述已自动生成

图5.1 ARMv8缓存层次结构

* 1. **缓存术语与ARMv8结构**

在传统的冯·诺伊曼架构下，指令和数据共享一个统一缓存。相比之下，修改后的哈佛架构采用独立的指令缓存(I-cache)和数据缓存(D-cache)，分别存储指令和数据。在现代ARMv8处理器中，常见的设计是使用不同的指令和数据L1缓存，以及一个共享的L2缓存。

缓存的基本构成包括地址、数据和状态信息。

标签(tag)用于标识与主内存地址相关的缓存行，决定缓存数据来源。缓存行是最小的加载单元，包含有效的数据或指令时被认为是有效，反之则无效。状态位用于指示数据是否可用，而在数据缓存中，脏位（dirty bit）可能用来标记缓存中数据是否与主内存不同需更新。

索引(index)决定地址映射到哪个缓存行，组(set)由共享相同索引的多个路组成。底部偏移量不需存储在标签中，因为最低位总是0。

图示

描述已自动生成

图5.2 缓存的基本构成

ARM核心中的主缓存一般采用组相联结构，不同于直接映射缓存，有效降低了缓存抖动的风险，提高了程序执行效率并带来更稳定的执行特性。这种设计增加了硬件复杂度，稍微提升功耗，因为每个周期需要比较多个标签。

组相联缓存将缓存分成多个相等大小的块，称为路（way）。内存位置可以映射到一路而非单一行，索引字段用于选定特定行，但指向每路内的一行。通常，L1数据缓存会有2路或4路，而Cortex-A57则拥有一个3路的L1指令缓存，L2缓存通常为16路。外部L3缓存如ARM CCN-504缓存一致性网络可具有更高关联度，因其规模更大。缓存行被分组到类似索引的缓存线集中，以验证命中需查看每个标签。

增加缓存关联度可减少抖动风险，全相联缓存是理想状态，但对于小缓存不切实际。实践中，8路以上关联度提升的性能改进有限，而16路关联对较大的L2缓存更具效益。

**5.3 缓存策略**

缓存策略决定了数据如何被加载到缓存中，以及如何在缓存中更新数据。ARMv8支持多种缓存策略，包括写分配（Write Allocation）、读分配（Read Allocation）、写回（Write-Back）和写直通（Write-Through）。这些策略影响着缓存的写入行为和内存的一致性维护。

写分配策略（Write Allocation）和读分配策略（Read Allocation）决定了何时将数据行分配到缓存中以及在存储指令执行时如何处理缓存中的命中与未命中情况。在写分配策略下，当发生写未命中时，会将相关的数据行加载到缓存中，这意味着在执行存储指令之前可能需要进行一次额外的读取操作。相比之下，读分配策略则会在读缺失时分配缓存行。这些策略的选择直接影响着处理器对于数据的访问效率和性能表现。

写回策略（Write-Back）和写直通策略（Write-Through）是两种常见的缓存更新方式。写回策略（Write-Back）会在写操作时只更新缓存而不立即更新外部内存，并标记相应的数据行为“脏”；

图示

描述已自动生成

图5.3 写回策略

写直通策略（Write-Through）则在写操作时同时将数据更新到缓存和外部内存系统中，保持缓存与外部内存的一致性。

图示

描述已自动生成

图5.4 写直通策略

ARMv8-A提供预加载提示指令。处理器具有预测访问内存的能力，可以自动将数据预取到缓存中，而无需程序员显式请求。预加载提示指令（Prefetch Hints）如PRFM PLDL1KEEP等可用于指示处理器预取特定数据，以加速后续的访问操作。

**5.4 缓存维护**

在软件系统中，清除缓存或使缓存无效是常见操作，尤其在外部内存发生更改、MMU权限变化以及动态生成代码需与缓存同步时需要执行。无效ating cache lines通过清除缓存行的有效位来实现数据清除，确保缓存与外部管理的内存保持一致。这些操作包括将标记为“脏”的数据写回下一级存储、清除脏标记以保持一致性。

清除缓存或缓存行的目的在于确保缓存行的内容与下层存储一致，这一过程适用于使用写回策略的数据缓存。清零操作则是将缓存中的内存块置零，无需访问外部内存，提高效率。在AArch64架构中，缓存维护指令通过指定操作和地址来执行，满足数据一致性和性能需求。

AArch64缓存维护操作是通过以下指令执行的:

<cache> <operation>{, <Xt>}

有许多操作可用。

表格

中度可信度描述已自动生成

图5.5 缓存操作

这些操作的执行点可基于虚拟地址或缓存组进行定义。维护指令可以按程序顺序执行，除非通过DSB指令明确排序。在多核处理器环境下，指定地址的数据缓存操作需按顺序执行，而其他操作相对于它们可并行执行。清零操作可有效利用缓存，更快地执行代码。

**5.5 缓存架构总结**

ARMv8架构的缓存设计是其高性能计算能力的关键。通过精心设计的缓存层次结构、一致性协议、以及灵活的缓存策略，ARMv8能够在保证数据一致性的同时，提供快速的数据处理能力。缓存维护和发现机制进一步增强了软件对硬件的控制能力，使得开发者能够更好地优化其应用程序，以适应不同的计算需求。

# 五、 小 结

综上所述，ARMv8架构在多个方面进行了创新和优化，使其在性能和能效上达到了新的高度。浮点运算单元和NEON技术的改进，使得ARMv8在多媒体处理和科学计算等高计算需求的应用中表现出色。内存管理单元通过有效的虚拟地址到物理地址转换，支持了系统的多任务处理能力。流水线结构的优化和缓存设计的改进，进一步提升了处理器的整体性能和能效。ARMv8架构凭借其优越的设计，成为现代计算领域中不可或缺的重要组成部分。未来，随着应用需求的不断增加，ARMv8将继续发挥其独特的优势，推动科技进步和行业发展。

**参考文献**

1. Ryzhyk, Leonid. "The arm architecture." Chicago University, Illinois, EUA (2006).
2. 戴维·A.帕特森，约翰·L.亨尼斯.计算机组成与设计：硬件/软件接口(原书第5版·RISC-V版).北京：机械工业出版社，2020：238-240.
3. ARM Cortex-A Series: Programmer’s Guide for ARMv8-A
4. 《微型计算机》2005年 第3期 123-123页 共1页