PM0214
Programming manual

STM32 Cortex®-M4 MCUs and MPUs programming manual

介绍

本编程手册为应用级和系统级软件开发人员提供信息。它提供了STM32 Cortex®-M4处理器编程模型、指令集和核心外设的完整说明。适用的产品列表如下表所示。

用于STM32F3系列、STM32F4系列、STM32G4系列、STM32H745/755和STM32H747/757系列、STM32L4系列、STM32L4+系列的Cortex®-M4处理器
STM32WB系列、STM32WL系列和STM32MP1系列是面向微控制器和微处理器市场的高性能32位处理器。它为开发者提供了诸多优势，包括:

- 卓越的处理性能结合快速的中断处理
- 增强的系统调试功能，具备全面的断点和跟踪能力
- 高效处理器核心、系统和内存
- 超低功耗，集成睡眠模式
- 平台安全

表 1. 适用产品

Type	Product Series and Lines
Microcontrollers	STM32F3 Series, STM32F4 Series, STM32G4 Series, STM32L4 Series, STM32L4+ Series, STM32WB Series, STM32WL Series STM32H745/755 and STM32H747/757 Lines
Microprocessors	STM32MP1 Series

参考文档

可在STMicroelectronics网站获取：*www.st.com*

- 数据手册 of STM32F3系列, STM32F4系列, STM32G4系列, STM32H745/755和STM32H747/757产品线, STM32L4系列, STM32L4+系列, STM32MP1系列, STM32WB系列和STM32WL系列
- STM32F3系列，STM32F4系列，STM32G4系列，STM32H745/755和STM32H747/757产品线，STM32L4系列，STM32L4+系列，STM32MP1系列，STM32WB系列和STM32WL系列

目录

表格列表

图录

1关于此文档

本文档提供了应用和系统级软件开发所需的信息。它不提供有关调试组件、功能或操作的信息。

本材料适用于微控制器软件和硬件工程师，包括那些没有Arm产品经验的工程师。

本文件适用于基于 Arm®(a) 的设备。

1.1 排版约定

本文件中使用的排版约定如下：

*italic*突出显示重要注释，引入特殊术语，表示内部交叉引用和引用。<和> 用于汇编语法中的可替换项，当它们出现在代码或代码片段中时。例如： LDRSB<cond> <Rt>, [<Rn>, #<offset>] 粗体用于突出显示界面元素，如菜单名称。表示信号名称。也用于描述列表中的适当术语。 等宽字体表示可在键盘上输入的文本，如命令、文件名和程序名。等宽字体表示命令或选项的允许缩写。您可以输入下划线文本代替完整的命令或选项名称。等宽斜体用于表示作为等宽文本参数的特定值。

等宽加粗用于表示语言关键字，当其用于示例代码之外的场景时。

1.2 寄存器缩写列表

以下缩写用于寄存器描述中：

a„Arm 是 a r
读/写 (rw) 软件可以读写这些位。只读 (r) 软件只能读取这些位
。只写 (w) 软件只能写入这个位。 读取该位将返回复位值。

egistered trademark of Arm Limited (or its subsidiaries) in the US and或者在其他地方。

读/清 (rc_w1) 软件可以通过写入 1 来读取和清除此位。 写入 ‘0’ 对位值无影响。读/清 (rc_w0) 软件可以通过写入 0 来读取和清除此位。 写入 ‘1’ 对位值无影响。翻转 (t) 软件只能通过写入 ‘1’ 来翻转此位。写入 ‘0’ 无影响。保留 (Res.) 保留位，必须保持复位值。

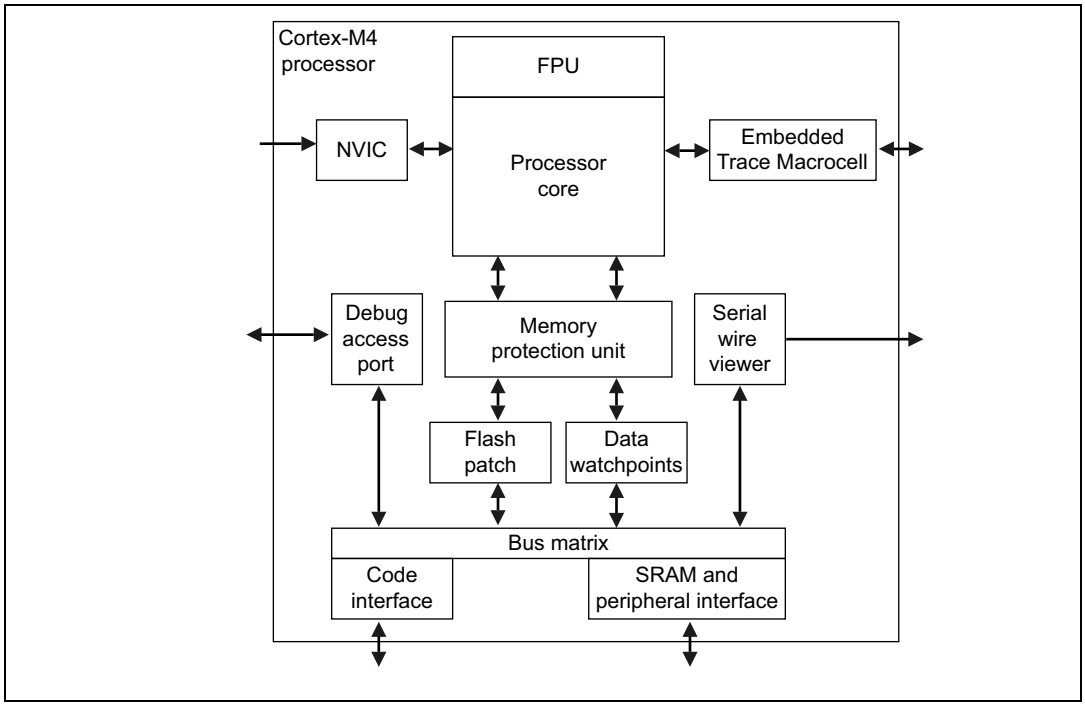
1.3 关于STM32 Cortex-M4处理器及核心外设

Cortex-M4处理器是一款专为微控制器市场设计的高性能32位处理器。它为开发者提供了诸多优势，包括:

- 卓越的处理性能结合快速的中断处理
- 增强的系统调试 具有全面的断点和跟踪功能
- 高效处理器核心，系统和内存
- 超低功耗，集成睡眠模式
- 平台安全鲁棒性，内置内存保护单元 (MPU)。

Cortex-M4处理器基于高性能处理器核心，采用3级流水线哈佛架构，使其非常适合高要求的嵌入式应用。该处理器通过高效的指令集和高度优化的设计实现卓越的功耗效率，提供高端处理硬件，包括IEEE754-兼容的单精度浮点运算、单周期和SIMD乘法、乘加功能、饱和算术以及专用硬件除法。

图1. STM32 Cortex-M4实现



为便于设计成本敏感的设备，Cortex-M4处理器实现了紧密耦合的系统组件，从而在减少处理器面积的同时显著提升中断处理和系统调试能力。Cortex-M4处理器实现了基于Thumb-2技术的Thumb®指令集版本，确保高代码密度和降低的程序内存需求。Cortex-M4指令集提供了现代32位架构所预期的卓越性能，具有8位和16位微控制器的高代码密度。

Cortex-M4处理器紧密集成了一个可配置的嵌套中断控制器（NVIC），以实现行业领先的中断性能。NVIC包含一个不可屏蔽中断（NMI），并提供高达256个中断优先级级别。处理器核心与NVIC的紧密集成可实现中断服务程序（ISRs）的快速执行，显著降低中断延迟。这是通过寄存器的硬件堆栈以及挂起加载-存储操作的能力实现的。中断处理程序不需要任何汇编程序存根，从而消除ISRs中的代码开销。尾部链接优化也显著降低了在切换中断服务程序时的开销。

为了优化低功耗设计，集成在NVIC中的睡眠模式中的深睡眠功能使STM32能够进入Stop或Standby模式。

1.3.1 系统级接口

Cortex-M4处理器采用AMBA®技术提供多种接口，以实现高速、低延迟的内存访问。它支持非对齐数据访问，并实现原子位操作，从而实现更快的外设控制、系统自旋锁和线程安全的布尔数据处理。

Cortex-M4处理器配备了一个内存保护单元（MPU），提供细粒度内存控制，使应用程序能够利用多个特权级别，按任务划分的方式分离和保护代码、数据和堆栈。这些要求在许多嵌入式应用中至关重要，例如汽车电子。

1.3.2 集成的可配置调试

Cortex-M4处理器实现了完整的硬件调试方案。这通过传统JTAG接口或一个2针*Serial Wire Debug*（SWD）接口，为处理器和内存提供了高系统可见性，特别适用于小型封装设备。

对于系统跟踪，处理器集成了一个*Instrumentation Trace Macrocell*（ITM），以及数据断点和一个剖析单元。为了实现简单且经济高效的系统事件性能分析，一个*Serial Wire Viewer*（SWV）可以通过单个引脚导出软件生成的消息、数据跟踪和性能分析信息的流。

可选的 {v*} (ETM) 在面积远小于传统跟踪单元的情况下，提供了无与伦比的指令跟踪捕获。

1.3.3 公司 Rtex-M4处理器特点和优势 su 摘要

- 紧密集成系统外设可减少面积和开发成本。
- Thumb指令集结合高代码密度与32位性能
- 符合IEEE 754标准的单精度浮点运算单元，集成于所有STM32 Cortex-M4微控制器中
- 系统组件的功率控制优化
- 集成睡眠模式以降低功耗
- 快速代码执行允许更慢的处理器时钟或增加睡眠模式时间
- 硬件除法和快速乘法器
- 确定性、高性能的中断处理用于时间关键型应用
- 用于安全关键应用的内存保护单元 {v*}
- 强大的调试和跟踪功能：Serial Wire Debug 和 Serial Wire Trace 可减少调试和跟踪所需的引脚数量。

1.3.4 Cortex-M4核心外设

外设包括:

嵌套向量中断控制器

nested vectored interrupt controller（NVIC）是一个嵌入式中断控制器，它支持低延迟中断处理。

系统控制块

The *system control block* (SCB) 是处理器的程序员模型接口。它提供系统实现信息和系统控制，包括系统异常的配置、控制和报告。

系统定时器

系统定时器（SysTick）是一个24位倒计时定时器。可以将其用作实时操作系统（RTOS）的节拍定时器或作为简单计数器。

内存保护单元

该 *Memory protection unit*（MPU）通过定义不同内存区域的内存属性来提高系统可靠性。它提供多达八个不同的区域，以及一个可选的预定义背景区域。

浮点单元

The *Floating-point unit* (FPU) 提供符合 IEEE754 标准的单精度、32 位浮点数值运算。

的 Cortex-M4 处理器

2.1 编程模型

本节描述了Cortex-M4编程器模型。除了各个核心寄存器的描述，还包含有关处理器模式和特权级别的信息，用于软件执行和堆栈。

2.1.1 处理器模式和软件执行的特权级别

处理器 *modes* 为:

线程模式：用于执行应用程序软件. 处理器在复位后进入线程模式. CONTROL寄存器控制软件执行是否为特权或非特权，参见 *CONTROL register on page 25*。

处理模式：用于处理异常。 当处理器完成异常处理后，会返回到线程模式。 软件执行始终是特权的。

软件执行的*privilege levels*是:

非特权级：*Unprivileged software* 在非特权级执行，并:

- 对MSR和MRS指令具有有限的访问权限，且无法使用CPS指令。
- 无法访问系统定时器、NVIC 或系统控制块。
- 可能对内存或外设的访问进行了限制。
- 必须使用SVC指令来生成一个*supervisor call*以转移控制到特权软件。

特权的: *Privileged software*在特权级别执行，并且可以使用所有指令，可以访问所有资源。 可以向CONTROL寄存器写入以更改软件执行的特权级别。

2.1.2 栈

处理器使用一个全降栈。这意味着栈指针指示栈内存中最后压入的项。当处理器将新项压入栈时，它会将栈指针递减，然后将该项写入新的内存位置。处理器实现两个栈，*main stack*和*process stack*，它们具有独立的栈指针副本，参见*Stack pointer on page 19*。

在线程模式下，CONTROL寄存器控制处理器使用主堆栈还是进程堆栈，参见*CONTROL register on page 25*。在处理模式下，处理器始终使用主堆栈。处理器操作的选项包括:

处理器模式、执行特权级别、an 的数组 d 堆栈使用情况

Processor mode	Used to execute	Privilege level for software execution	Stack used
Thread	Applications	Privileged or unprivileged(1)	Main stack or process stack(1)
Handler	Exception handlers	Always privileged	Main stack

1. 参见 *CONTROL register on page 25*。

2.1.3 核心寄存器

图2. 处理器核心寄存器

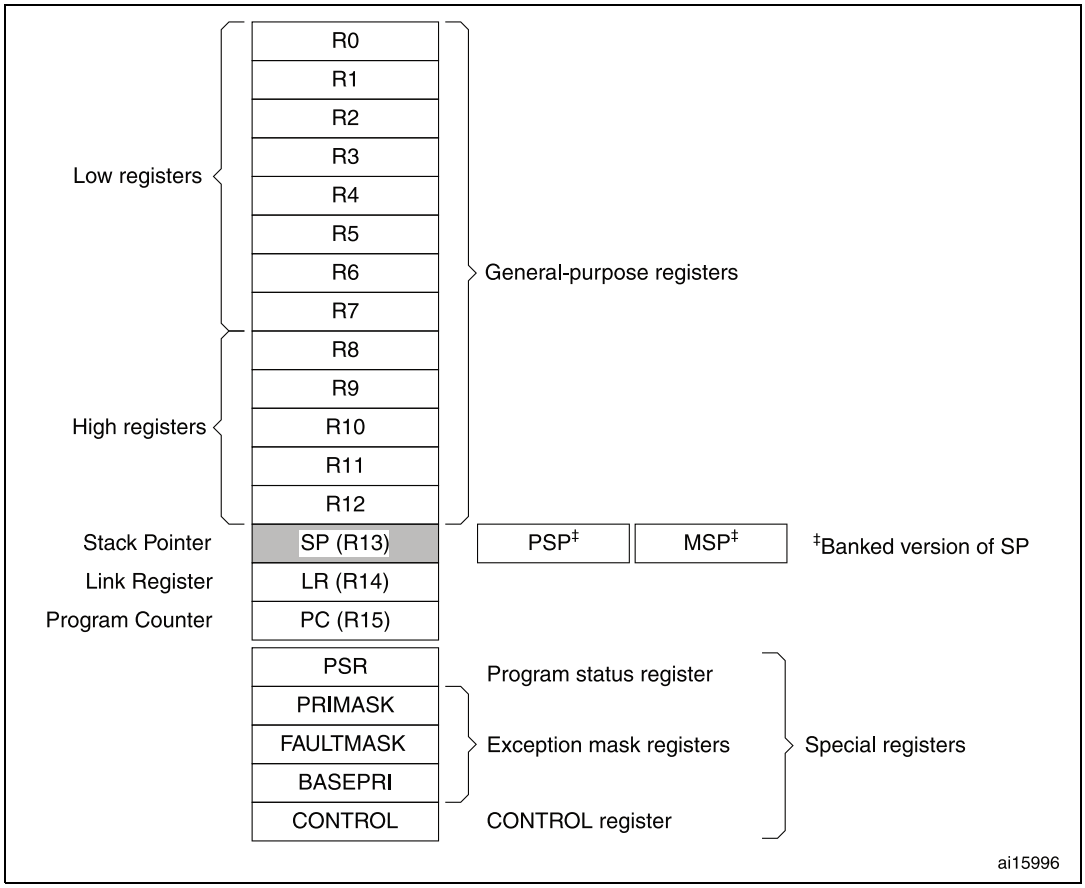


表2. 总结 表 3. 核心寄存器组摘要

Name	Type(1)	Required privilege(2)	Reset value	Description
R0-R12	read-write	Either	Unknown	*General-purpose registers on page 19*
MSP	read-write	Privileged	See description	*Stack pointer on page 19*
PSP	read-write	Either	Unknown	*Stack pointer on page 19*
LR	read-write	Either	0xFFFFFFFF	*Link register on page 19*
PC	read-write	Either	See description	*Program counter on page 19*

表 3. 核心寄存器集合摘要（续）

Name	Type(1)	Required privilege(2)	Reset value	Description
PSR	read-write	Privileged	0x01000000	*Program status register on page 19*
ASPR	read-write	Either	Unknown	*Application program status register on page 21*
IPSR	read-only	Privileged	0x00000000	*Interrupt program status register on page 22*
EPSR	read-only	Privileged	0x01000000	*Execution program status register on page 22*
PRIMASK	read-write	Privileged	0x00000000	*Priority mask register on page 24*
FAULTMASK	read-write	Privileged	0x00000000	*Fault mask register on page 24*
BASEPRI	read-write	Privileged	0x00000000	*Base priority mask register on page 25*
CONTROL	read-write	Privileged	0x00000000	*CONTROL register on page 25*

1. 描述在程序执行期间线程模式和处理模式下的访问类型。调试访问可能有所不同。2. 无论是哪种入口，特权软件和非特权软件均可访问该寄存器。

通用寄存器

R0-R12 是用于数据操作的 32 位通用寄存器。

堆栈指针

The *Stack Pointer* (SP) is register R13. In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

- 0: *Main Stack Pointer* (MSP)。这是复位值。
- 1: *Process Stack Pointer* (PSP)

复位时，p 处理器从地址0x加载MSP的值 00000000

链接寄存器

Link Register（LR）是寄存器R14。它用于存储子程序、函数调用和异常的返回信息。上电复位时，处理器会加载LR值0xFFFFFFFF。

程序计数器

Program Counter（PC）是寄存器R15。它包含当前程序地址。在复位时，处理器将PC加载为复位向量的值，该值位于地址0x00000004处。在复位时，该值的位[0]会被加载到EPSR的T位，且必须为1。

程序状态寄存器

该 *Program Status Register*（PSR）结合：

- *Application Program Status Register*（APSR）
- *Interrupt Program Status Register*（IPSR）
- *Execution Program Status Register*(EPSR)

这些寄存器是32位PSR中的互斥位字段。位分配如*Figure 3*和*Figure 4*所示。

图3. APSR、IPSR和EPSR位分配

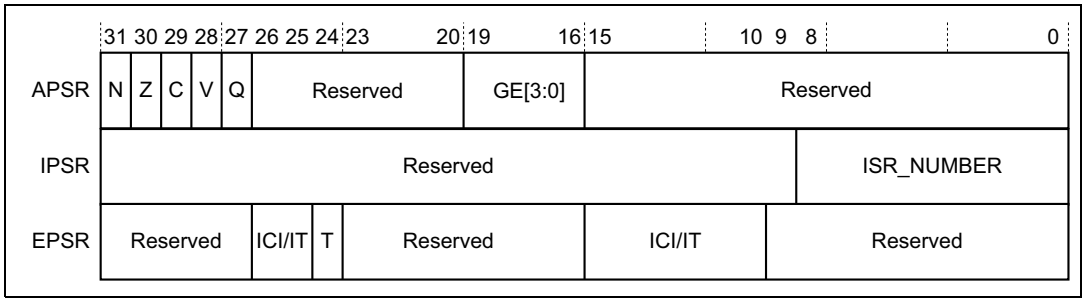
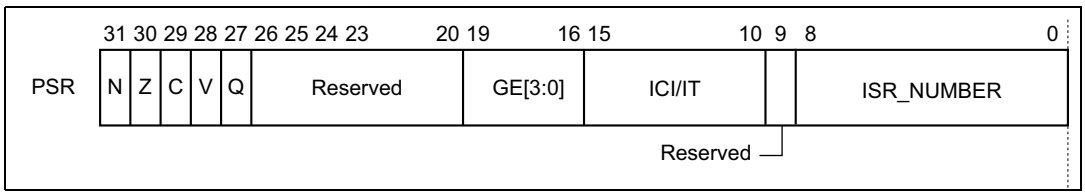


图4. PSR位分配



通过将寄存器名称作为MSR或MRS指令的参数，可以单独访问这些寄存器，或以任意两个或全部三个寄存器的组合方式访问。例如：

- 使用MRS指令读取所有PSR寄存器。
- 使用 MSR 指令通过 APSR_nzcvq 写入 APSR 的 N、Z、C、V 和 Q 位。

PSR的组合和属性如下：

表4. PSR寄存器组合

Register	Type	Combination
PSR	read-write(1), (2)	APSR, EPSR, and IPSR
IEPSR	read-only	EPSR and IPSR
IAPSR	read-write(1)	APSR and IPSR
EAPSR	read-write(2)	APSR and EPSR

1. 处理器忽略对IPSR位的写入。 2. 读取EPSR位返回零，处理器忽略对这些位的写入。

参见指令描述 *MRS on page 186* 和 *MSR on page 187* 以获取有关如何访问程序状态寄存器的更多信息。

应用程序状态寄存器

APSR 包含前一条指令执行的条件标志的当前状态。请参阅 *Table 3 on page 18* 中的寄存器摘要以了解其属性。位分配如下：

表5. 应用程序状态寄存器位定义

Bits	Description
Bit 31	**N:** Negative or less than flag: 0: Operation result was positive, zero, greater than, or equal 1: Operation result was negative or less than.
Bit 30	**Z:** Zero flag: 0: Operation result was not zero 1: Operation result was zero.
Bit 29	**C:** Carry or borrow flag: 0: Add operation did not result in a carry bit or subtract operation resulted in a borrow bit 1: Add operation resulted in a carry bit or subtract operation did not result in a borrow bit.
Bit 28	**V:** Overflow flag: 0: Operation did not result in an overflow 1: Operation resulted in an overflow.
Bit 27	**Q:** DSP overflow and saturation flag: Sticky saturation flag. 0: Indicates that saturation has not occurred since reset or since the bit was last cleared to zero 1: Indicates when an SSAT or USAT instruction results in saturation, or indicates a DSP overflow. This bit is cleared to zero by software using an MRS instruction.
Bits 26:20	Reserved.
Bits 19:16	**GE[3:0]:** Greater than or Equal flags. See *SEL on page 105* for more information.
Bits 15:0	Reserved.

中断程序状态寄存器

IPSR 包含当前 *Interrupt Service Routine*（ISR）的异常类型号。参见 *Table 3 on page 18* 中的寄存器摘要以了解其属性。 位分配如下：

表6. IPSR位定义

Bits	Description
Bits 31:9	Reserved
Bits 8:0	**ISR_NUMBER**: This is the number of the current exception: 0: Thread mode 1: Reserved 2: NMI 3: Hard fault 4: Memory management fault 5: Bus fault 6: Usage fault 7: Reserved .... 10: Reserved 11: SVCall 12: Reserved for Debug 13: Reserved 14: PendSV 15: SysTick 16: IRQ0(1) .... 255: IRQ240(1) see *Exception types on page 37* for more information.

1. 取决于具体产品。请参阅相关STM32产品的参考手册/数据手册以获取相关信息。

执行程序状态寄存器

EPSR 包含 Thumb 状态位，以及用于任一的执行状态位：

- *If-Then*(IT) 指令
- *Interruptible-Continuable Instruction*(ICI) 字段用于中断的加载多个或存储多个指令。

请参阅 *Table 3 on page 18* 中的寄存器摘要以获取 EPSR 属性。位分配如下：

表7. EPSR位定义

Bits	Description
Bits 31:27	Reserved.
Bits 26:25, 15:10	**ICI**: Interruptible-continuable instruction bits, see *Interruptible-continuable instructions on page 23*.
Bits 26:25, 15:10	**IT**: Indicates the execution state bits of the IT instruction, see *IT on page 145*.
Bit 24	T: Thumb state bit.
Bits 23:16	Reserved.
Bits 9:0	Reserved.

尝试通过应用程序软件直接读取EPSR始终返回零。尝试通过应用程序软件中的MSR指令写入EPSR将被忽略。故障处理程序可以检查堆叠的PSR中的EPSR值以指示发生故障的操作。参见*Section 2.3.7: Exception entry and return on page 42*。

可中断可继续的指令

当在执行LDM STM、PUSH、POP、VLDM、VSTM、VPUSH或VPOP指令时发生中断，处理器:

- 暂时停止加载多个或存储多个指令操作
- 将多操作中的下一个寄存器操作数存储到EPSR的位[15:12]中。

处理完中断后，处理器:

- 返回到由 bits[15:12] 所指向的寄存器
- 恢复执行多个加载或存储指令。

当EPSR处于ICI执行状态时，位[26:25,11:10]为零。

如果-那么块

If-Then块包含最多四条指令，这些指令跟随一个16位IT指令。块中的每条指令都是条件性的。指令的条件要么全部相同，要么其中一些可以是其他条件的相反。更多信息请参见*IT on page 145*。

拇指状态

Cortex-M4处理器仅支持在Thumb状态下的指令执行。以下操作可以将T位清零:

- 指令 BLX, BX 和 POP{PC}
- 从异常返回时堆栈中的{xPSR}值恢复
- 位[0]的向量值在异常入口或复位时

当T位为0时尝试执行指令会导致故障或锁死。详见 *Lockup on page 47*。

异常屏蔽寄存器

异常屏蔽寄存器用于禁用处理器对异常的处理。禁用可能影响时间关键任务的异常。

要访问异常屏蔽寄存器，可以使用MSR和MRS指令，或使用CPS指令修改PRIMASK或FAULTMASK的值。如需更多信息，请参见*MRS on page 186*、*MSR on page 187*和*CPS on page 182*。

优先级屏蔽寄存器

PRIMASK寄存器可防止所有具有可配置优先级的异常的触发。请参见*Table 3 on page 18*中的寄存器摘要以了解其属性。*Figure 5*展示了位分配。

图5. PRIMASK位分配

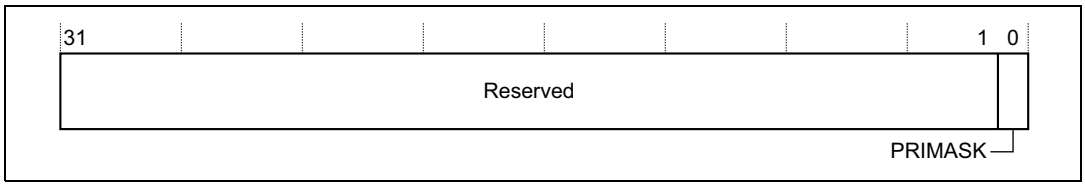


表 8. PRIMASK 寄存器位定义

Bits	Description
Bits 31:1	Reserved
Bit 0	**PRIMASK**: 0: No effect 1: Prevents the activation of all exceptions with configurable priority.

故障屏蔽寄存器

FAULTMASK寄存器防止触发所有异常，除了*Non-Maskable Interrupt*（NMI）。请参阅*Table 3 on page 18*中的寄存器摘要以了解其属性。*Figure 6*显示位分配。

图6. FAULTMASK位分配

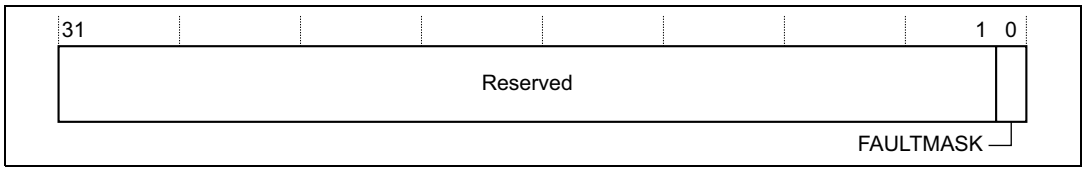


表9. FAULTMASK寄存器位定义

Bits	Function
Bits 31:1	Reserved
Bit 0	**FAULTMASK**: 0: No effect 1: Prevents the activation of all exceptions except for NMI.

处理器在退出任何异常处理程序（除非屏蔽中断处理程序外）时，将FAULTMASK位清除为0。

基础优先级掩码寄存器

BASEPRI寄存器定义异常处理的最低优先级。当BASEPRI设置为非零值时，会阻止所有优先级相同或低于BASEPRI值的异常的激活。其属性请参见*Table 3 on page 18*中的寄存器摘要。*Figure 7*显示位分配。

图7. BASEPRI位分配

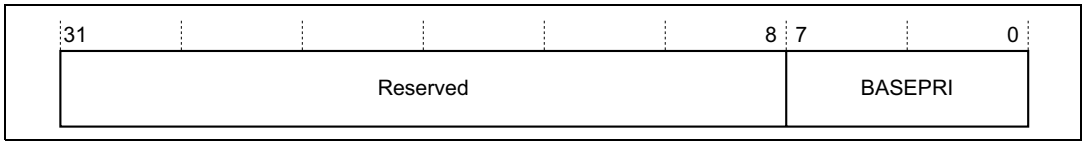


表 10. BASEPRI寄存器位分配

Bits	Function
Bits 31:8	Reserved
Bits 7:4	**BASEPRI[7:4]** Priority mask bits(1) 0x00: no effect Nonzero: defines the base priority for exception processing. The processor does not process any exception with a priority value greater than or equal to BASEPRI.
Bits 3:0	Reserved

1. 该字段与中断优先级寄存器中的优先级字段类似。请参阅 *Interrupt priority register x (NVIC_IPRx) on page 215* 了解更多详情。请注意，较高的优先级字段值对应较低的异常优先级。

控制寄存器

CONTROL寄存器控制处理器处于线程模式时软件执行所使用的堆栈和特权级别，并指示FPU状态是否处于活动状态。其属性请参见*Table 3 on page 18*中的寄存器摘要。

表 11. CONTROL 寄存器位定义

Bits	Function
Bits 31:3	Reserved
Bit 2	**FPCA:** Indicates whether floating-point context currently active: 0: No floating-point context active 1: Floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
Bit 1	**SPSEL:** Active stack pointer selection. Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return.
Bit 0	**nPRIV:** Thread mode privilege level. Defines the Thread mode privilege level. 0: Privileged 1: Unprivileged.

处理模式始终使用{v*}，因此当处于处理模式时，处理器会忽略对控制寄存器活动栈指针位的显式写入。异常入口和返回机制会更新控制寄存器。

在操作系统环境中，建议在Thread模式下运行的线程使用进程栈，内核和异常处理程序使用主栈。

默认情况下，线程模式使用MSP。要将线程模式中使用的堆栈指针切换到PSP，可以:

- 使用MSR指令将活动堆栈指针位设置为1，参见*MSR on page 187*。
- 执行异常返回到线程模式的操作，使用适当的 EXC_RETURN 值，参见 *Exception return behavior on page 44*。

当更改堆栈指针时，软件必须在MSR指令之后立即使用ISB指令。这确保了在ISB之后执行的指令使用新的堆栈指针。参见 *ISB on page 185*

2.1.4 异常和中断

Cortex-M4 处理器支持中断和系统异常。处理器和 *Nested Vectored Interrupt Controller*（NVIC）对所有异常进行优先处理并进行处理。异常会改变软件控制的正常流程。如需更多信息，请参阅 *Exception entry on page 42*、*Exception* 和 *return on page 44*。

NVIC寄存器控制中断处理。参见*Nested vectored interrupt controller (NVIC) on page 208*以获取更多信息。

2.1.5 数据类型

处理器:

- 支持以下数据类型:
  – 32位字 – 16位半字
  – 8位字节

- 管理所有内存访问作为小端序。参考 *Memory regions, types and attributes on page 29* 以获取更多信息。

2.1.6 Cortex微控制器软件接口标准 (CMSIS)

对于一个Cortex-M4微控制器系统，*Cortex Microcontroller Software Interface Standard*(CMSIS) 定义了:

- 一种常见的方式是: – 访问外设
  寄存器 – 定义异常向量

- 以下名称: – 核心外设的寄存器 – 核心异
  常向量

- 一个设备无关的实时操作系统内核接口，包括调试通道

CMSIS 包括 Cortex-M4 处理器中核心外设的地址定义和数据结构。

CMSIS 通过启用模板代码的复用以及从各种中间件供应商处组合符合 CMSIS 标准的软件组件，简化了软件开发。软件供应商可以扩展 CMSIS 以包含其外设定义和这些外设的访问函数。

本文档包含CMSIS定义的寄存器名称，并提供针对处理器核心和核心外设的CMSIS函数的简要描述。

本文档使用由CMSIS定义的寄存器简称。在某些情况下，这些与可能在其他文档中使用的架构简称有所不同。

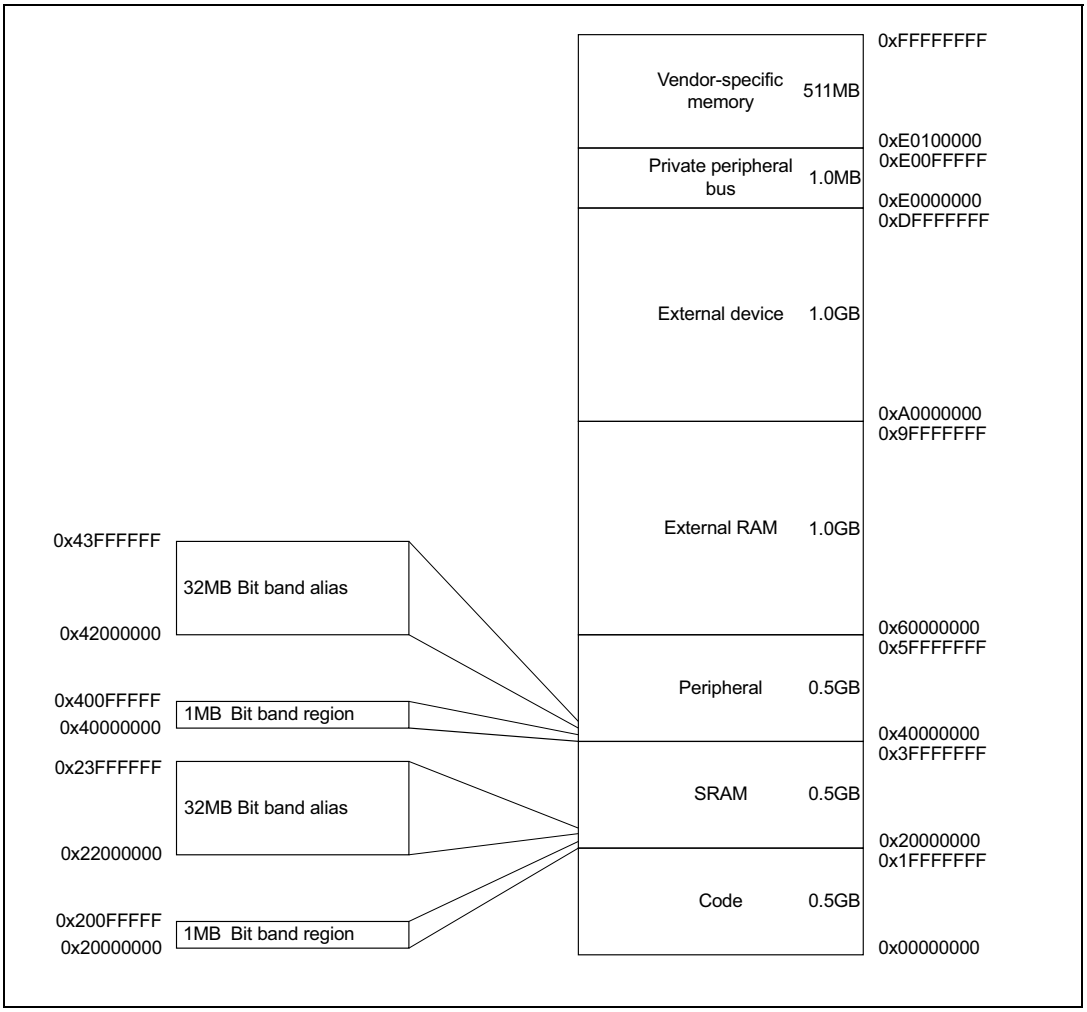
以下部分提供了有关CMSIS的更多信息：

- *Section 2.5.4: Power management programming hints on page 49*
- *CMSIS intrinsic functions on page 58*
- *Interrupt set-enable register x (NVIC_ISERx) on page 210*
- *NVIC programming hints on page 218*

2.2 内存模型

本节描述了处理器的内存映射、内存访问行为以及位带功能。处理器具有固定的内存映射，提供最多4 GB的可寻址内存。

图8. 内存映射



SRAM 和外设区域包括位带区域。位带技术为位数据提供了原子操作，参见 *Section 2.2.5: Bit-banding on page 32*。

处理器保留*Private peripheral bus*（PPB）地址范围中的区域用于核心外设寄存器，参见 *Section 4.1: About the STM32 Cortex-M4 core peripherals on page 193*。

2.2.1 内存区域、类型和属性 utes

内存映射和MPU编程将内存映射划分为区域。每个区域都有定义的内存类型，某些区域还有额外的内存属性。内存类型和属性决定了对该区域的访问行为。{v*}

内存类型是:

正常	处理器可以为了提高效率重新排序事务，或进行推测性读取。
设备	处理器保持事务顺序相对于其他事务，针对设备或强序内存。
强序的	处理器保持事务顺序相对于所有其他事务。

不同的排序要求对于设备内存和强序内存意味着内存系统可以缓冲对设备内存的写入，但不得缓冲对强序内存的写入。

额外的内存属性包括: {v*}

Execute Never（XN）	表示处理器阻止指令访问。任何尝试从XN区域获取指令都会导致内存管理故障异常。

2.2.2 内存系统内存访问顺序

大多数由显式内存访问指令引起的内存访问，内存系统不能保证这些访问完成的顺序与指令的程序顺序一致，只要这不会影响指令序列的行为。通常，如果正确的程序执行依赖于两个内存访问按程序顺序完成，软件必须在内存访问指令之间插入内存屏障指令，参见 *Section 2.2.4: Software ordering of memory accesses on page 31*。

然而，内存系统确实保证对设备和强序内存的访问具有某种顺序。对于两个内存访问指令A1和A2，如果A1在程序顺序中出现在A2之前，则由这两个指令引起的内存访问顺序为:

表 12. 内存访问顺序(1)

A1	A2			
	Normal access	Device access		Strongly ordered access
		Non-shareable	Shareable	
Normal access	-	-	-	-
Device access, non-shareable	-	<	-	<
Device access, shareable	-	-	<	<
Strongly ordered access	-	<	<	<

1. - 表示内存系统不保证访问的顺序。 < 表示访问按程序顺序观察，即 A1 始终在 A2 之前被观察。

2.2.3 内存访问的行为

内存映射中每个区域的访问行为是：

表13. 内存访问行为

Address range	Memory region	Memory type	XN	Description
0x00000000-0x1FFFFFFF	Code	Normal(1)	-	Executable region for program code. Can also put data here.
0x20000000-0x3FFFFFFF	SRAM	Normal(1)	-	Executable region for data. Can also put code here. This region includes bit band and bit band alias areas, see *Table 14 on page 32*.
0x40000000-0x5FFFFFFF	Peripheral	Device(1)	XN(1)	This region includes bit band and bit band alias areas, see *Table 15 on page 32*.
0x60000000-0x9FFFFFFF	External RAM	Normal(1)	-	Executable region for data.
0xA0000000-0xDFFFFFFF	External device	Device(1)	XN(1)	External Device memory
0xED000000-0xED0FFFFF	Private Peripheral Bus	Strongly-ordered(1)	XN(1)	This region includes the NVIC, system timer, and system control block.
0xED100000-0xFFFFFFFF	Memory mapped peripherals	Device(1)	XN(1)	This region includes all the STM32 standard peripherals.

1. 参见 *Memory regions, types and attributes on page 29* 以获取更多信息。

代码、静态随机存取存储器和外部RAM区域可以存储程序。然而，建议程序始终使用代码区域。其原因是处理器拥有独立的总线，能够同时进行指令获取和数据访问。

MPU 可以覆盖本节中描述的默认内存访问行为。如需更多信息，请参阅 *Memory protection unit (MPU) on page 193*。

指令预取与分支预测

Cortex-M4处理器：

- 在执行前预取指令
- 推测性地从分支目标地址预取。

2.2.4 软件控制的内存访问顺序

程序流程中的指令顺序并不总是能保证对应的内存事务顺序。原因在于:

- 处理器可以重新排序某些内存访问以提高效率，前提是这不会影响指令序列的行为。
- 处理器具有多个总线接口。
- 内存或内存映射中的设备具有不同的等待状态。
- 一些内存访问是缓冲的或推测的。

*Section 2.2.2: Memory system ordering of memory accesses on page 29*描述了内存系统保证内存访问顺序的情况。否则，如果内存访问顺序是关键的，软件必须包含内存屏障指令以强制该顺序。处理器提供以下内存屏障指令:

DMB 该 *Data Memory Barrier* (DMB) 指令确保未完成的内存事务在后续内存事务之前完成。参见 *DMB on page 183*。DSB 该 *Data Synchronization Barrier* (DSB) 指令确保未完成的内存事务在后续指令执行之前完成。参见 *DSB on page 184*。ISB 该 *Instruction Synchronization Barrier* (ISB) 确保所有已完成的内存事务的效果对后续指令可见。参见 *ISB on page 185*。

在例如中使用内存屏障指令:

- 向量表。如果程序修改了向量表中的某个条目，然后使能相应的异常，请在操作之间使用DMB指令。这确保了如果在使能后立即触发异常，处理器将使用新的异常向量。
- 自修改代码。如果程序包含自修改代码，应在程序中的代码修改后立即使用ISB指令。这确保了后续指令执行使用更新后的程序。
- 内存映射切换。如果系统包含内存映射切换机制，在程序中切换内存映射后，使用DSB指令。这确保了后续指令的执行使用更新后的内存映射。
- 动态异常优先级更改。当异常处于待处理或激活状态时需要更改异常优先级，应在更改后使用DSB指令。这确保了在完成DSB指令后更改生效。
- 在多主系统中使用信号量。如果系统包含多个总线主设备，例如，如果系统中存在另一个处理器，每个处理器必须在任何信号量指令之后使用DMB指令，以确保其他总线主设备看到内存事务的执行顺序。

对强序内存的内存访问，例如系统控制块，不需要使用DMB指令。

在内存保护单元编程时，使用DSB指令后接ISB指令或异常返回，以确保后续指令使用新的内存保护单元配置。

2.2.5 位带

位带区域将每个 *bit-band alias*区域中的字映射到位带区域的*bit-band region*中的单个位。位带区域占据了SRAM和外围存储器区域中最低的1 Mbyte。

内存映射包含两个32 Mbyte的别名区域，这些区域映射到两个1 Mbyte的位带区域：

- 对32 Mbyte SRAM别名区域的访问映射到1 Mbyte SRAM位带区域，如*Table 14*所示。
- 对32 MB外设别名区域的访问映射到1 MB外设位带区域，如*Table 15*所示。

表14. SRAM存储器位带区域

Address range	Memory region	Instruction and data accesses
0x20000000-0x200FFFFF	SRAM bit-band region	Direct accesses to this memory range behave as SRAM memory accesses, but this region is also bit addressable through bit-band alias.
0x22000000-0x23FFFFFF	SRAM bit-band alias	Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write. Instruction accesses are not remapped.

表15. 外设存储器位带区域

Address range	Memory region	Instruction and data accesses
0x40000000-0x400FFFFF	Peripheral bit-band region	Direct accesses to this memory range behave as peripheral memory accesses, but this region is also bit addressable through bit-band alias.
0x42000000-0x43FFFFFF	Peripheral bit-band alias	Data accesses to this region are remapped to bit-band region. A write operation is performed as read-modify-write. Instruction accesses are not permitted.

Note: A word access to the SRAM or peripheral bit-band alias regions map to a single bit in the SRAM or peripheral bit-band region.

Bit band accesses can use byte, halfword, or word transfers. The bit band transfer size matches the transfer size of the instruction making the bit band access.

以下公式展示了别名区域如何映射到位带区域：

位字偏移量 = (字节偏移量 x 32) + (位数 x 4)

位字地址 = 位带基地址 + 位字偏移量

其中：

- {v*} 是指目标位在位带内存区域中的位置。
- Bit_word_addr 是别名内存区域中映射到目标位的字的地址。

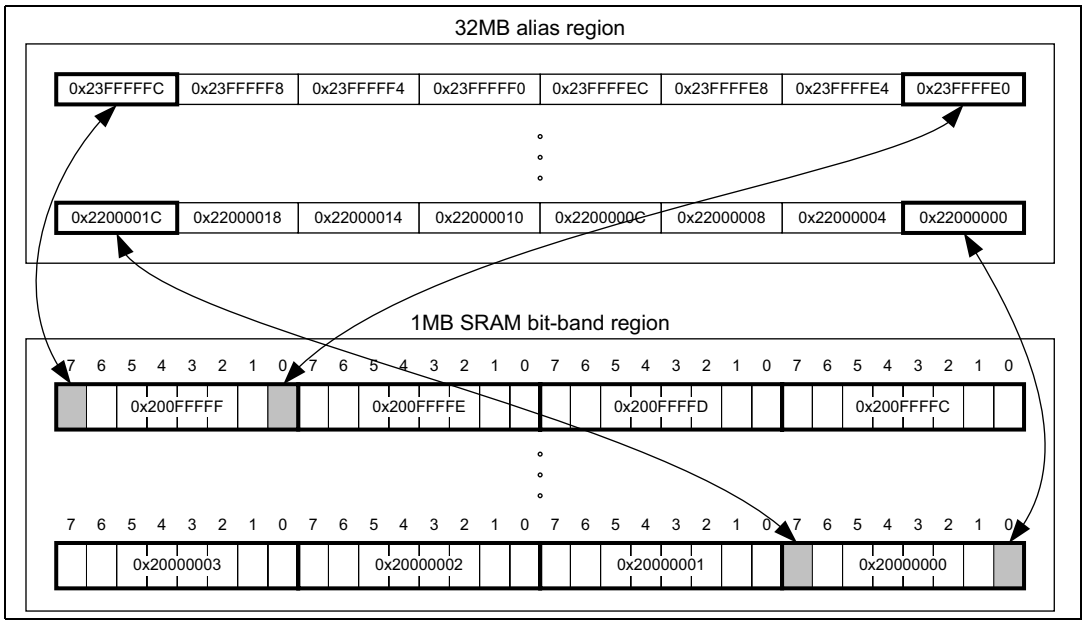
- 位带基地址是别名区域的起始地址。
- 字偏移量是指位带区域内包含目标位的字的编号。

- Bit_number 是目标位的位位置，范围为 0-7。

*Figure 9 on page 33*展示SRAM位带别名区域与SRAM位带区域之间的位带映射示例：

- 位于0x23FFFFED处的别名字映射到位带字节的位[0]处 0x200FFFFF: 0x23FFFFED = 0x22000000 + (0xFFFFF*32) + (0*4).
- 地址0x23FFFFFC处的别名字映射到 地址0x200FFFFF处的位带字节的bit[7]：0x23FFFFFC = 0x22000000 + (0xFFFFF*32) + (7*4)。
- 位于0x22000000的别名字映射到位带字节 0x20000000的bit[0]：0x22000000 = 0x22000000 + (0*32) + (0 *4)。
- 位于0x2200001C的别名字映射到位于 0x20000000的位带字节的bit[7]：0x2200001C = 0x22000000+ (0*32) + (7*4)。

图9. 位带映射



直接访问别名区域

在别名区域中的一个字写入会更新位带区域中的一个比特。

写入别名区域中一个字的值的Bit[0]决定了写入位带区域中目标位的值。将bit[0]置为1的值写入时，会将1写入位带位；将bit[0]置为0的值写入时，会将0写入位带位。

别名字的Bits[31:1]位对位带位没有影响。写入0x01与写入0xFF效果相同。写入0x00与写入0x0E效果相同。

在别名区域中读取一个字：

- 0x00000000 表示位带区域中的目标位被设置为零
- 0x00000001 表示位带区域中的目标位被置为 1

直接访问位带区域

*Behavior of memory accesses on page 30*描述直接的字节、半字或字访问到位带区域的行为。

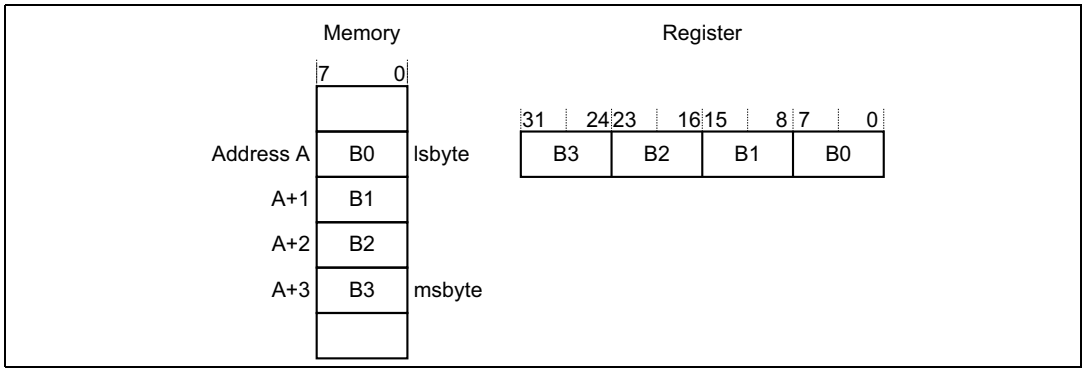
2.2.6 内存字节序

处理器将内存视为一个按从零开始递增的顺序编号的字节线性集合。例如，字节0-3存储了第一个字，字节4-7存储了第二个字。

小端格式

在小端格式中，处理器将一个字的最低有效字节存储在最低编号的字节中，最高有效字节存储在最高编号的字节中。参见 *Figure 10* 以获取示例。

图10. 小端序示例



2.2.7 同步原语

Cortex-M4指令集包含*synchronization primitives*对。这些提供一种非阻塞机制，线程或进程可以使用该机制以获得对内存位置的独占访问权限。软件可以使用它们来执行保证的读-修改-写内存更新序列，或用于信号量机制。

一对同步原语包括：

- 加载独占指令：用于读取内存位置的值，请求对该位置的独占访问。
- 存储独占指令：用于尝试写入同一内存位置，并返回一个状态位到寄存器。如果该位为0：线程或进程获得了对内存的独占访问权限，且写入成功。如果该位为1：线程或进程未获得对内存的独占访问权限，且未执行写入。

加载独占和存储独占指令对 s 是:

- 这些指令的名称 LDREX 和 STREX
- 半字指令 LDREXH 和 STREXH
- 字节指令 LDREXB 和 STREXB。

软件必须使用独占加载指令和相应的独占存储指令。

为了执行一个内存位置的原子读取-修改-写入操作，软件必须:
1. 使用加载独占指令读取该位置的值。2. 根据需要更新该值。3. 使用存储独占指令尝试将新值写回内存位置。4. 测试返回的状态位。如果该位是:

0: 读取-修改-写入操作已成功完成。
1: 未执行写操作。这表明步骤1返回的值可能已过时。软件必须重试{v*}序列。

软件可以使用同步原语来实现信号量如下:
1. 使用加载独占指令从信号量地址读取以检查信号量是否可用。2. 如果信号量可用，使用存储独占指令将占用值写入信号量地址。3. 如果步骤2返回的状态位表示存储独占指令成功则

软件已声称信号量。然而，如果Store-Exclusive失败，软件执行步骤1之后，另一个进程可能已经声称了信号量。

Cortex-M4 包含一个独占访问监视器，该监视器标记处理器已执行加载独占指令的事实。如果处理器属于多处理器系统，系统还会全局标记每个处理器的独占访问所针对的内存地址。

处理器在以下情况下会移除其独占访问标记:

- 它执行一条CLREX指令。
- 它执行一条Store-Exclusive指令，无论写操作是否成功。
- 发生异常。这意味着处理器可以解决不同线程之间的信号量冲突。

在一个多处理器实现中，执行一个:

- CLREX指令仅清除处理器的本地独占访问标签。
- 存储独占指令或异常会移除处理器的本地独占访问标记和全局独占访问标记。

如需了解有关同步原语指令的更多信息，请参见 *LDREX and STREX on page 79* 和 *CLREX on page 80*。

1 第

2.2.8 Prog 关于同步原语的强制提示 积极的{v*}

ISO/IEC C 无法直接生成独占访问指令。CMSIS 提供用于生成这些指令的内联函数:

表16. CMSIS函数用于独占访问指令

Instruction	CMSIS function
LDREX	`uint32_t __LDREXW (uint32_t *addr)`
LDREXH	`uint16_t __LDREXH (uint16_t *addr)`
LDREXB	`uint8_t __LDREXB (uint8_t *addr)`
STREX	`uint32_t __STREXW (uint32_t value, uint32_t *addr)`
STREXH	`uint32_t __STREXH (uint16_t value, uint16_t *addr)`
STREXB	`uint32_t __STREXB (uint8_t value, uint8_t *addr)`
CLREX	`void __CLREX (void)`

例如:

uint16_t value; uint16_t *address = 0x20001002; value = __LDREXH (address); // 从内存地址加载16位值 //0x20001002

2.3 异常模型

This section describes the exception model.

2.3.1 Exception states

Each exception is in one of the following states:

Inactive The exception is not active and not pending.

Pending The exception is waiting to be serviced by the processor. An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.

Active An exception that is being serviced by the processor but has not completed.

Note: An exception handler can interrupt the execution of another exception handler. In this case both exceptions are in the active state.

Active and pending The exception is being serviced by the processor and there is a pending exception from the same source.

2.3.2 Exception types

The exception types are:

Reset Reset is invoked on power up or a warm reset. The exception model treats reset as a special form of exception. When reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When reset is deasserted, execution restarts from the address provided by the reset entry in the vector table. Execution restarts as privileged execution in Thread mode.

NMI A *NonMaskable Interrupt* (NMI) can be signalled by a peripheral or triggered by software. This is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2. NMIs cannot be:

- Masked or prevented from activation by any other exception
- Preempted by any exception other than Reset.

Hard fault A hard fault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism. Hard faults have a fixed priority of -1, meaning they have higher priority than any exception with configurable priority.

Memory management fault A memory management fault is an exception that occurs because of a memory protection related fault. The MPU or the fixed memory protection constraints determines this fault, for both instruction and data memory transactions. This fault is used to abort instruction accesses to *Execute Never* (XN) memory regions.

Bus fault A bus fault is an exception that occurs because of a memory related fault for an instruction or data memory transaction. This might be from an error detected on a bus in the memory system.

Usage fault A usage fault is an exception that occurs in case of an instruction execution fault. This includes:

- An undefined instruction
- An illegal unaligned access
- Invalid state on instruction execution
- An error on exception return.

The following can cause a usage fault when the core is configured to report it:

- An unaligned address on word and halfword memory access
- Division by zero

SVCall A *supervisor call* (SVC) is an exception that is triggered by the SVC instruction. In an OS environment, applications can use SVC instructions to access OS kernel functions and device drivers.

PendSV PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.

SysTick A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.

Interrupt (IRQ) An interrupt, or IRQ, is an exception signalled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

表17. 不同异常类型的属性

Exception number(1)	IRQ number(1)	Exception type	Priority	Vector address or offset(2)	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	Hard fault	-1	0x0000000C	-
4	-12	Memory management fault	Configurable(3)	0x00000010	Synchronous
5	-11	Bus fault	Configurable(3)	0x00000014	Synchronous when precise Asynchronous when imprecise
6	-10	Usage fault	Configurable(3)	0x00000018	Synchronous
7-10	-	-	-	Reserved	-
11	-5	SVCall	Configurable(3)	0x0000002C	Synchronous
12-13	-	-	-	Reserved	-
14	-2	PendSV	Configurable(3)	0x00000038	Asynchronous

表格 17. 不同异常类型的属性 (con 继续的)

Exception number[1]	IRQ number[1]	Exception type	Priority	Vector address or offset[2]	Activation
15	-1	SysTick	Configurable[3]	0x0000003C	Asynchronous
16 and above	0 and above	Interrupt (IRQ)	Configurable[4]	0x00000040 and above[5]	Asynchronous

1. 为简化软件层，CMSIS 仅使用 IRQ 编号，因此除中断外的异常使用负值。IPSR 返回异常编号。更多信息请参见 *Interrupt program status register on page 22*。2. 更多信息请参见 *Vector table on page 40*。3. 请参见 *System handler priority registers (SHPRx) on page 233*。4. 请参见 *Interrupt priority register x (NVIC_IPRx) on page 215*。5. 以4步为单位递增。

对于除复位以外的异步异常，处理器可以在异常被触发和进入异常处理程序之间执行另一条指令。

特权软件可以禁用*Table 17 on page 38*显示为具有可配置优先级的异常。如需进一步信息，请参见：

- *System handler control and state register (SHCSR) on page 235*
- *Interrupt clear-enable register x (NVIC_ICERx) on page 211*

如需了解更多信息关于硬故障、内存管理故障、总线故障和使用故障，请参见 *Section 2.4: Fault handling on page 44*。

2.3.3 异常处理程序

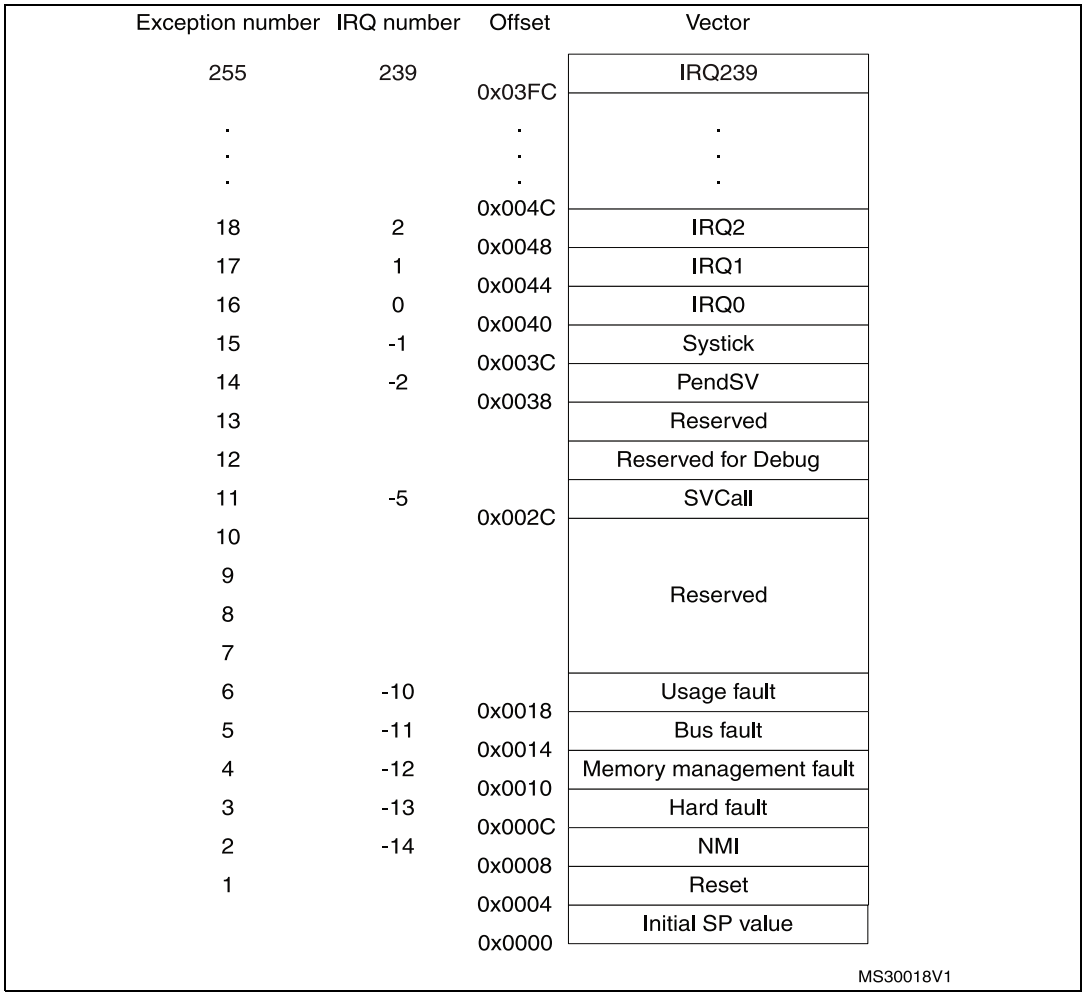
处理器通过以下方式处理异常：

中断服务程序（ISRs）	中断 IRQ0 到 IRQ81 是由中断服务程序处理的异常。
错误处理程序	硬故障、内存管理故障、使用故障、总线故障是由故障处理程序处理的故障异常。
系统处理程序	NMI、PendSV、SVCall、SysTick 以及异常故障都是由系统处理程序处理的系统异常。

2.3.4 向量表

向量表包含堆栈指针的复位值，以及所有异常处理程序的起始地址，也称为异常向量。*Figure 11 on page 40* 显示向量表中异常向量的顺序。每个向量的最低有效位必须为 1，表示异常处理程序是 Thumb 代码。

图11. 向量表



系统复位时，向量表固定在地址 0x00000000。特权软件可以写入 VTOR 以将向量表起始地址重新定位到不同的内存位置，范围为 0x00000080 到 0x3FFFFF80。如需进一步信息，请参见 *Vector table offset register (VTOR) on page 227*。

2.3.5 异常优先级

*Table 17 on page 38*表明所有异常都有一个关联的优先级，详情如下:

- 较低的优先级值表示较高的优先级
- 所有异常（除复位、硬故障和非屏蔽中断外）的可配置优先级。

如果软件未配置任何优先级，则所有具有可配置优先级的异常优先级为0。有关配置异常优先级的信息，请参见

- *System handler priority registers (SHPRx) on page 233*
- *Interrupt priority register x (NVIC_IPRx) on page 215*

可配置的优先级值范围为0-15。这意味着复位、硬故障和NMI异常具有固定的负优先级值，始终比任何其他异常具有更高的优先级。

例如，将 IRQ[0] 赋予更高的优先级值，将 IRQ[1] 赋予更低的优先级值，意味着 IRQ[1] 的优先级高于 IRQ[0]。如果 IRQ[1] 和 IRQ[0] 都被触发，IRQ[1] 会在 IRQ[0] 之前被处理。

如果有多个待处理异常具有相同的优先级，则异常编号较低的待处理异常优先处理。例如，如果IRQ[0]和IRQ[1]都处于待处理状态且具有相同的优先级，则IRQ[0]会在IRQ[1]之前被处理。

当处理器正在执行异常处理程序时，如果发生更高优先级的异常，该处理程序将被中断。如果发生与当前处理异常同优先级的异常，处理程序不会被中断，无论异常编号如何。然而，新中断的状态将变为待处理。

2.3.6 中断优先级分组

为了在具有中断的系统中提高优先级控制，{v*} 支持优先级分组。这将每个中断优先级寄存器条目划分为两个字段:

- 一个上字段定义了 *group priority*
- 一个下级字段，用于在组内定义 *subpriority*。

只有组优先级决定了中断异常的抢占。当处理器正在执行中断异常处理程序时，另一个具有与正在处理的中断相同组优先级的中断不会抢占处理程序，

如果多个待处理中断具有相同的组优先级，则子优先级字段决定了它们的处理顺序。如果多个待处理中断具有相同的组优先级和子优先级，则具有最低IRQ编号的中断将被优先处理。

有关将中断优先级字段分为组优先级和子优先级的信息，请参见
Application interrupt and reset control register (AIRCR) on page 228。

2.3.7 异常入口和返回

异常处理的描述使用以下术语:

P 当处理器正在执行异常处理程序时，如果另一个异常的优先级高于当前正在处理的异常的优先级，该异常可以抢占异常处理程序。有关中断抢占的更多信息，请参见 *Section 2.3.6: Interrupt priority grouping*。

当一个异常中断另一个异常时，这些异常被称为嵌套异常。更多信息请参见 *Exception entry on page 42*。

返回 这种情况发生在异常处理程序完成时，且:

- 没有待处理的异常具有足够的优先级需要处理
- 已完成的异常处理程序未能处理延迟到达的异常。

处理器弹出堆栈，并将处理器状态恢复到中断发生前的状态。参见 *Exception return on page 44* 以获取更多信息。

Tail-chaining 该机制加快异常处理。在异常处理程序执行完毕后，如果存在一个待处理异常，其满足异常入口的条件，则跳过栈弹出操作，并将控制转移到新的异常处理程序。

L迟到到达 这种机制加快了抢占。如果在处理先前异常的状态保存过程中发生更高优先级的异常，处理器会切换到处理该更高优先级的异常，并启动该异常的{v*}获取。状态保存不受迟到到达的影响，因为两种异常保存的状态相同。因此，状态保存可以不间断地继续。处理器可以在原始异常的异常处理程序第一条指令进入处理器执行阶段之前接受迟到到达的异常。在处理迟到到达的异常的异常处理程序返回后，正常的尾部链接规则适用。

异常条目

异常入口发生于存在具有足够高优先级的待处理异常时，或者:

- 处理器处于线程模式
- 新的异常具有比正在处理的异常更高的优先级，此时新的异常会抢占原始异常。

当一个异常中断另一个异常时，异常会被嵌套。

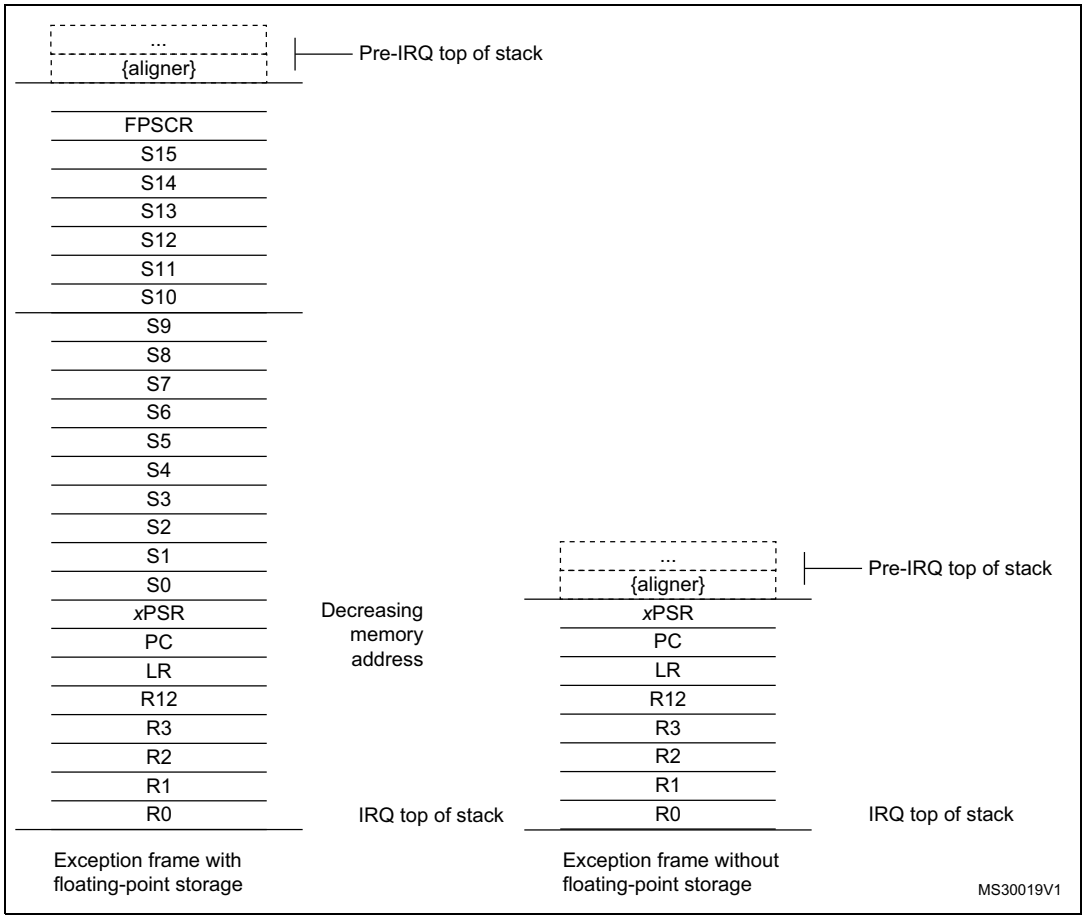
足够的优先级意味着该异常的优先级高于由屏蔽寄存器设置的任何限制。更多信息请参见 *Exception mask registers on page 23*。优先级低于此的异常虽处于待处理状态，但不会被处理器处理。

当处理器发生异常时，除非该异常是尾链异常或迟到异常，否则处理器会将信息压入当前栈。此操作称为 *stacking*，八字数据结构称为 *stack frame*。

当使用浮点运算例程时，Cortex-M4处理器会自动将架构的浮点状态压栈到异常进入时。*Figure 12 on page 43*展示了当浮点状态因中断或异常而保留在堆栈中的Cortex-M4堆栈框架布局。当未为浮点状态分配堆栈空间时，该

栈帧与无浮点单元的Armv7-M实现相同。*Figure 12 on page 43* 也展示此栈帧。

图12. Cortex-M4栈帧布局



在压栈之后立即，堆栈指针指向堆栈帧中的最低地址。堆栈帧的对齐通过配置控制寄存器（CCR）的STKALIGN位进行控制。

堆栈帧包含返回地址。这是被中断程序中下一条指令的地址。该值在异常返回时被恢复到程序计数器，以便被中断的程序继续执行。

与堆栈操作同时进行，处理器执行向量获取，从向量表中读取异常处理程序起始地址。当堆栈操作完成时，处理器开始执行异常处理程序。同时，处理器将EXC_RETURN值写入LR。这表示哪个栈指针对应于栈帧，以及处理器在进入发生前所处的操作模式。

如果在异常进入期间没有更高优先级的异常发生，处理器开始执行异常处理程序，并自动将相应的待处理中断状态改为激活状态。

如果在进入异常处理程序时发生另一个更高优先级的异常，处理器将开始执行该异常的异常处理程序，并且不改变先前异常的待处理状态。这种情况称为晚到异常。

异常返回

当处理器处于处理模式并执行以下任一指令以将EXC_RETURN值加载到PC中时，会发生异常返回。

- 一条加载PC的LDM或POP指令
- 一条以程序计数器为目的地的LDR指令
- 使用任何寄存器的BX指令。

EXC_RETURN 是在异常进入时加载到 LR 中的值。异常机制依赖此值来检测处理器是否已完成异常处理程序。此值的最低五位提供了有关返回栈和处理器模式的信息。*Table 18* 显示了 EXC_RETURN 值及其异常返回行为的描述。

所有EXC_RETURN值的位[31:5]均被置为1。当此值被加载到程序计数器时，它会指示处理器异常已处理完毕，并启动相应的异常返回序列。

表 18. 异常返回行为

EXC_RETURN[31:0]	Description
0xFFFFFFF1	Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return.
0xFFFFFFE1	Return to Handler mode, exception return uses floating-point-state from MSP and execution uses MSP after return.
0xFFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.

2.4 故障处理

错误是异常的一个子集。如需了解更多信息，请参见 *Exception model on page 37*。
以下元素生成错误：

- 总线错误发生于：– 指令获取或向量表加载
  – 数据访问
- 内部检测到的错误，例如未定义的指令
- 尝试执行一条指令，从一个标记为 *Non-Executable* 的内存区域 (XN)。
- 特权违规或尝试访问未管理区域导致的MPU故障。

2.4.1 故障类型

*Table 19*显示故障类型、用于处理故障的处理程序、对应的故障状态寄存器以及表示故障已发生的寄存器位。如需了解有关故障状态寄存器的更多信息，请参见 *Configurable fault status register (CFSR; UFSR+BFSR+MMFSR) on page 237*。

表格 19. 故障

Fault	Handler	Bit name	Fault status register
Bus error on a vector read	Hard fault	VECTTBL	*Hard fault status register (HFSR) on page 241*
Fault escalated to a hard fault		FORCED	
MPU or default memory map mismatch:	MemManage	-	*Memory management fault address register (MMFAR) on page 242*
– on instruction access		IACCVIOL(1)	
– on data access		DACCVIOL	
– during exception stacking		MSTKERR	
– during exception unstacking		MUNSKERR	
– during lazy floating-point state preservation		MLSPERR	
Bus error:	Bus fault	-	-
– During exception stacking		STKERR	*Bus fault address register (BFAR) on page 242*
– During exception unstacking		UNSTKERR	
– During instruction prefetch		IBUSERR	
– During lazy floating-point state preservation		LSPERR	
Precise data bus error		PRECISERR	
Imprecise data bus error		IMPRECISERR	
Attempt to access a coprocessor	Usage fault	NOCP	*Configurable fault status register (CFSR; UFSR+BFSR+MMFSR) on page 237*
Undefined instruction		UNDEFINSTR	
Attempt to enter an invalid instruction set state(2)		INVSTATE	
Invalid EXC_RETURN value		INVPC	
Illegal unaligned load or store		UNALIGNED	
Divide By 0		DIVBYZERO	

1. 即使MPU被禁用，在访问XN区域时仍会发生。

2. 尝试使用除Thumb指令集以外的指令集，或返回到带有ICI延续的非加载/存储多指令。

2.4.2 故障升级和硬故障

所有故障异常（除硬故障外）都有可配置的异常优先级，如*System handler priority registers (SHPRx) on page 233*所述。软件可以禁用这些故障处理程序的执行，如*System handler control and state register (SHCSR) on page 235*所述。

通常，异常优先级，以及异常屏蔽寄存器的值，决定处理器是否进入故障处理程序，以及故障处理程序能否抢占另一个故障处理程序，如*Section 2.3: Exception model on page 37*所述。

在某些情况下，具有可配置优先级的故障被视为硬故障。这被称为 *priority escalation*，故障被描述为 *escalated to hard fault*。升级为硬故障的情况包括：

- 故障处理程序会导致与它处理的相同类型的故障。这种升级到硬故障的情况发生在故障处理程序无法抢占自身时，因为它必须具有与当前优先级级别相同的优先级。
- 当故障处理程序处理故障时，它会引发一个具有相同或更低优先级的故障。这是因为新故障的处理程序无法抢占当前正在执行的故障处理程序。
- 一个异常处理程序导致一个故障，其优先级与当前正在执行的异常相同或更低。
- 发生故障，且其处理程序未启用。

当进入总线故障处理程序时，如果在堆栈压入过程中发生总线故障，总线故障不会升级为硬故障。这意味着如果损坏的堆栈导致故障，即使处理程序的堆栈压入失败，故障处理程序仍会执行。故障处理程序运行，但堆栈内容已损坏。

只有Reset和NMI可以抢占固定优先级的硬故障。硬故障可以抢占除Reset、NMI或另一个硬故障之外的任何异常。

2.4.3 F 故障状态寄存器和故障地址寄存器 简洁的

故障状态寄存器指示故障的原因。对于总线故障和内存管理故障，故障地址寄存器指示导致故障的操作所访问的地址，如*Table 20*所示。

表20. 故障状态和故障地址寄存器

Handler	Status register name	Address register name	Register description
Hard fault	HFSR	-	*Hard fault status register (HFSR) on page 241*
Memory management fault	MMFSR	MMFAR	*Memory management fault address register (MMFAR) on page 242*
Bus fault	BFSR	BFAR	*Bus fault address register (BFAR) on page 242*
Usage fault	UFSR	-	*Configurable fault status register (CFSR; UFSR+BFSR+MMFSR) on page 237*

2.4.4 锁死

如果在执行不可屏蔽中断或硬故障处理程序时发生硬故障，处理器将进入锁定状态。当处理器处于锁定状态时，它不会执行任何指令。处理器将保持锁定状态，直到以下任一情况发生：

- 它被重置了
- 发生非屏蔽中断
- 它被调试器暂停

如果锁死状态由NMI处理程序引发，后续的NMI不会导致处理器退出锁死状态。

2.5 电源管理

STM32和Cortex-M4处理器的睡眠模式降低功耗：

- 睡眠模式停止处理器时钟。所有其他系统和外围时钟可能仍在运行。
- 深度睡眠模式停止了STM32系统和外设时钟的大部分功能。在产品级别，这对应于停止模式或待机模式。如需更多详情，请参阅STM32参考手册中的**电源模式**章节。

SCR中的SLEEPDEEP位用于选择使用的睡眠模式，如*System control register (SCR) on page 230*所述。如需了解睡眠模式的行为，请参阅STM32产品参考手册。

本节描述进入睡眠模式的机制，以及从睡眠模式唤醒的条件。

2.5.1 进入睡眠模式

本节描述了软件可以用来将处理器置于睡眠模式的机制。

系统可能会产生虚假的唤醒事件，例如调试操作唤醒处理器。因此，软件必须能够在发生此类事件后将处理器重新置于睡眠模式。一个程序可能包含一个空闲循环，以将处理器重新置于睡眠模式。

等待中断

*wait for interrupt*指令，WFI，会立即进入睡眠模式（除非唤醒条件为真，如 *Wakeup from WFI or sleep-on-exit on page 48*所示）。当处理器执行WFI指令时，它会停止执行指令并进入睡眠模式。更多信息请参见 *WFI on page 192*。

等待事件

*wait for event*指令WFE根据一位事件寄存器的值使处理器进入睡眠模式。当处理器执行WFE指令时，它会检查事件寄存器的值：

- 0: 处理器停止执行指令并进入睡眠模式
- 1: 处理器将寄存器清零，并继续执行指令，无需进入睡眠模式。

参见 *WFE on page 191* 以获取更多信息。

如果事件寄存器为1，这表示处理器在执行WFE指令时不得进入睡眠模式。通常，这是由于外部事件信号被激活，或系统中的某个处理器执行了SEV指令，如*SEV on page 189*所示。软件不能直接访问此寄存器。

退出时睡眠

如果SCR中的SLEEPONEXIT位被设置为1，当处理器完成异常处理程序的执行时，它返回到线程模式并立即进入睡眠模式。在仅需要处理器在异常发生时运行的应用程序中使用此机制。

2.5.2 从睡眠模式唤醒

处理器唤醒的条件取决于导致其进入睡眠模式的机制。

从WFI唤醒或退出时睡眠

通常，处理器只有在检测到具有足够优先级以导致异常进入的异常时才会唤醒。

某些嵌入式系统可能需要在处理器唤醒后、执行中断处理程序之前执行系统恢复任务。要实现此目的，请将PRIMASK位设置为1，将FAULTMASK位设置为0。如果到达的中断已启用且优先级高于当前异常优先级，处理器会唤醒但不会执行中断处理程序，直到处理器将PRIMASK设置为零。如需了解有关PRIMASK和FAULTMASK的更多信息，请参见 *Exception mask registers on page 23*。

从WFE唤醒

处理器在以下情况下唤醒:

- 它检测到具有足够优先级以导致异常入口的异常
- 它检测外部事件信号，参见 *Section 2.5.3: External event input / extended interrupt and event input*
- 在多处理器系统中，系统中的另一个处理器执行一个SEV指令。

此外，如果SCR中的SEVONPEND位被设置为1，则任何新的待处理中断都会触发一个事件并唤醒处理器，即使该中断被禁用或优先级不足，无法导致异常进入。如需了解有关SCR的更多信息，请参见*System control register (SCR) on page 230*。

2.5.3 外部事件输入 / 扩展的中断和事件输入

处理器提供外部事件输入信号。{v*}

该信号由外部或扩展中断/事件控制器（EXTI）在异步事件检测（来自外部输入引脚或异步外设事件）时生成。

此信号可以唤醒处理器从WFE，或设置内部WFE事件寄存器为1以指示处理器在后续的WFE指令中不得进入睡眠模式，如*Wait for event on page 48*所述。如需更多详细信息，请参阅STM32参考手册中的低功耗模式章节。

2.5.4 电源管理编程提示

ISO/IEC C不能直接生成WFI和WFE指令。CMSIS为这些指令提供了以下函数:

```
void __WFE(void) // 等待事件void __WFI(void) // 等
待中断
```

3 STM32 Cortex-M4 指令集

本章是用户指南中Cortex-M4指令集描述的参考资料。以下各节提供一般信息：

Section 3.1: Instruction set summary on page 50

Section 3.2: CMSIS intrinsic functions on page 58

Section 3.3: About the instruction descriptions on page 60

以下各节分别描述 Cortex-M4 指令的功能组。它们共同描述了 Cortex-M4 处理器支持的所有指令：

Section 3.4: Memory access instructions on page 69

Section 3.5: General data processing instructions on page 81

Section 3.6: Multiply and divide instructions on page 109

Section 3.7: Saturating instructions on page 125

Section 3.8: Packing and unpacking instructions on page 134

Section 3.9: Bitfield instructions on page 138

Section 3.10: Floating-point instructions on page 149

Section 3.11: Miscellaneous instructions on page 180

3.1 操作指令摘要

该处理器实现了一种Thumb指令集的版本。*Table 21* 列出了支持的指令。

在 *Table 21*：

- 尖括号，{v*}，用于括起操作数的替代形式。
- 大括号，{}，包含可选操作数。
- 操作数列不完整。
- Op2 是一个灵活的第二个操作数，可以是寄存器或常数。
- 大多数指令可以使用一个可选的条件码后缀。

如需关于指令和操作数的更多信息，请参见指令描述。

表21. Cortex-M4指令

Mnemonic	Operands	Brief description	Flags	Page
ADC, ADCS	{Rd,} Rn, Op2	Add with carry	N,Z,C,V	*3.5.1 on page 83*
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	*3.5.1 on page 83*
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V	*3.5.1 on page 83*
ADR	Rd, label	Load PC-relative address	—	*3.4.1 on page 70*

表21. Cortex-M4指令（续）

Mnemonic	Operands	Brief description	Flags	Page
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C	*3.5.2 on page 85*
ASR, ASRS	Rd, Rm, <Rs|#n>	Arithmetic shift right	N,Z,C	*3.5.3 on page 86*
B	label	Branch	—	*3.9.5 on page 142*
BFC	Rd, #lsb, #width	Bit field clear	—	*3.9.1 on page 139*
BFI	Rd, Rn, #lsb, #width	Bit field insert	—	*3.9.1 on page 139*
BIC, BICS	{Rd,} Rn, Op2	Bit clear	N,Z,C	*3.5.2 on page 85*
BKPT	#imm	Breakpoint	—	*3.11.1 on page 181*
BL	label	Branch with link	—	*3.9.5 on page 142*
BLX	Rm	Branch indirect with link	—	*3.9.5 on page 142*
BX	Rm	Branch indirect	—	*3.9.5 on page 142*
CBNZ	Rn, label	Compare and branch if non zero	—	*3.9.6 on page 144*
CBZ	Rn, label	Compare and branch if zero	—	*3.9.6 on page 144*
CLREX	—	Clear exclusive	—	*3.4.9 on page 80*
CLZ	Rd, Rm	Count leading zeros	—	*3.5.4 on page 87*
CMN	Rn, Op2	Compare negative	N,Z,C,V	*3.5.5 on page 88*
CMP	Rn, Op2	Compare	N,Z,C,V	*3.5.5 on page 88*
CPSID	iflags	Change processor state, disable interrupts	—	*3.11.2 on page 182*
CPSIE	iflags	Change processor state, enable interrupts	—	*3.11.2 on page 182*
DMB	—	Data memory barrier	—	*3.11.4 on page 184*
DSB	—	Data synchronization barrier	—	*3.11.4 on page 184*
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C	*3.5.2 on page 85*
ISB	—	Instruction synchronization barrier	—	*3.11.5 on page 185*
IT	—	If-then condition block	—	*3.9.7 on page 145*
LDM	Rn{!}, reglist	Load multiple registers, increment after	—	*3.4.6 on page 76*
LDMDB, LDMEA	Rn{!}, reglist	Load multiple registers, decrement before	—	*3.4.6 on page 76*
LDMFD, LDMIA	Rn{!}, reglist	Load multiple registers, increment after	—	*3.4.6 on page 76*
LDR	Rt, [Rn, #offset]	Load register with word	—	*3.4 on page 69*
LDRB, LDRBT	Rt, [Rn, #offset]	Load register with byte	—	*3.4 on page 69*
LDRD	Rt, Rt2, [Rn, #offset]	Load register with two bytes	—	*3.4.2 on page 71*
LDREX	Rt, [Rn, #offset]	Load register exclusive	—	*3.4.8 on page 79*

表21. Cortex-M4指令（续）

Mnemonic	Operands	Brief description	Flags	Page
LDREXB	Rt, [Rn]	Load register exclusive with byte	—	*3.4.8 on page 79*
LDREXH	Rt, [Rn]	Load register exclusive with halfword	—	*3.4.8 on page 79*
LDRH, LDRHT	Rt, [Rn, #offset]	Load register with halfword	—	*3.4 on page 69*
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load register with signed byte	—	*3.4 on page 69*
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load register with signed halfword	—	*3.4 on page 69*
LDRT	Rt, [Rn, #offset]	Load register with word	—	*3.4 on page 69*
LSL, LSLS	Rd, Rm, <Rs|#n>	Logical shift left	N,Z,C	*3.5.3 on page 86*
LSR, LSRS	Rd, Rm, <Rs|#n>	Logical shift right	N,Z,C	*3.5.3 on page 86*
MLA	Rd, Rn, Rm, Ra	Multiply with accumulate, 32-bit result	—	*3.6.1 on page 110*
MLS	Rd, Rn, Rm, Ra	Multiply and subtract, 32-bit result	—	*3.6.1 on page 110*
MOV, MOVS	Rd, Op2	Move	N,Z,C	*3.5.6 on page 89*
MOVT	Rd, #imm16	Move top	—	*3.5.7 on page 91*
MOVW, MOV	Rd, #imm16	Move 16-bit constant	N,Z,C	*3.5.6 on page 89*
MRS	Rd, spec_reg	Move from special register to general register	—	*3.11.6 on page 186*
MSR	spec_reg, Rm	Move from general register to special register	N,Z,C,V	*3.11.7 on page 187*
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z	*3.6.1 on page 110*
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C	*3.5.6 on page 89*
NOP	—	No operation	—	*3.11.8 on page 188*
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C	*3.5.2 on page 85*
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C	*3.5.2 on page 85*
PKHTB, PKHBT	{Rd,} Rn, Rm, Op2	Pack Halfword	-	*3.8.1 on page 135*
POP	reglist	Pop registers from stack	—	*3.4.7 on page 78*
PUSH	reglist	Push registers onto stack	—	*3.4.7 on page 78*
QADD	{Rd,} Rn, Rm	Saturating double and add	-	*3.7.3 on page 128*
QADD16	{Rd,} Rn, Rm	Saturating add 16	-	*3.7.3 on page 128*
QADD8	{Rd,} Rn, Rm	Saturating add 8	-	*3.7.3 on page 128*
QASX	{Rd,} Rn, Rm	Saturating add and subtract with exchange	-	*3.7.4 on page 129*

表21. Cortex-M4指令（续）

Mnemonic	Operands	Brief description	Flags	Page
QDADD	{Rd,} Rn, Rm	Saturating add	-	*3.7.5 on page 130*
QDSUB	{Rd,} Rn, Rm	Saturating double and subtract	-	*3.7.5 on page 130*
QSAX	{Rd,} Rn, Rm	Saturating subtract and add with exchange	-	*3.7.4 on page 129*
QSUB	{Rd,} Rn, Rm	Saturating subtract	-	*3.7.3 on page 128*
QSUB16	{Rd,} Rn, Rm	Saturating subtract 16	-	*3.7.4 on page 129*
QSUB8	{Rd,} Rn, Rm	Saturating subtract 8	-	*3.7.4 on page 129*
RBIT	Rd, Rn	Reverse bits	—	*3.7.4 on page 129*
REV	Rd, Rn	Reverse byte order in a word	—	*3.5.8 on page 92*
REV16	Rd, Rn	Reverse byte order in each halfword	—	*3.5.8 on page 92*
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	—	*3.5.8 on page 92*
ROR, RORS	Rd, Rm, <Rs|#n>	Rotate right	N,Z,C	*3.5.3 on page 86*
RRX, RRXS	Rd, Rm	Rotate right with extend	N,Z,C	*3.5.3 on page 86*
RSB, RSBS	{Rd,} Rn, Op2	Reverse subtract	N,Z,C,V	*3.5.1 on page 83*
SADD16	{Rd,} Rn, Rm	Signed add 16	-	*3.5.9 on page 93*
SADD8	{Rd,} Rn, Rm	Signed add 8	-	*3.5.9 on page 93*
SASX	{Rd,} Rn, Rm	Signed add and subtract with exchange	-	*3.5.14 on page 98*
SBC, SBCS	{Rd,} Rn, Op2	Subtract with carry	N,Z,C,V	*3.5.1 on page 83*
SBFX	Rd, Rn, #lsb, #width	Signed bit field extract	—	*3.9.2 on page 140*
SDIV	{Rd,} Rn, Rm	Signed divide	—	*3.6.3 on page 112*
SEV	—	Send event	—	*3.11.9 on page 189*
SHADD16	{Rd,} Rn, Rm	Signed halving add 16	—	*3.5.10 on page 94*
SHADD8	{Rd,} Rn, Rm	Signed halving add 8	—	*3.5.10 on page 94*
SHASX	{Rd,} Rn, Rm	Signed halving add and subtract with exchange	—	*3.5.11 on page 95*
SHSAX	{Rd,} Rn, Rm	Signed halving subtract and add with exchange	—	*3.5.11 on page 95*
SHSUB16	{Rd,} Rn, Rm	Signed halving subtract 16	—	*3.5.12 on page 96*
SHSUB8	{Rd,} Rn, Rm	Signed halving subtract 8	—	*3.5.12 on page 96*
SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed multiply accumulate long (halfwords)	Q	*3.6.3 on page 112*
SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed multiply accumulate dual	Q	*3.6.4 on page 114*

表21. Cortex-M4指令（续）

Mnemonic	Operands	Brief description	Flags	Page
SMLAL	RdLo, RdHi, Rn, Rm	Signed multiply with accumulate (32 x 32 + 64), 64-bit result	—	*3.6.2 on page 111*
SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed multiply accumulate long, halfwords	—	*3.6.5 on page 115*
SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed multiply accumulate long dual	—	*3.6.5 on page 115*
SMLAWB, SMLAWT	Rd, Rn, Rm, Ra	Signed multiply accumulate, word by halfword	Q	*3.6.3 on page 112*
SMLSD	Rd, Rn, Rm, Ra	Signed multiply subtract dual	Q	*3.6.6 on page 117*
SMLSLD	RdLo, RdHi, Rn, Rm	Signed multiply subtract long dual	—	*3.6.6 on page 117*
SMMLA	Rd, Rn, Rm, Ra	Signed most significant word multiply accumulate	—	*3.6.7 on page 119*
SMMLS, SMMLR	Rd, Rn, Rm, Ra	Signed most significant word multiply subtract	—	*3.6.7 on page 119*
SMMUL, SMMULR	{Rd,} Rn, Rm	Signed most significant word multiply	—	*3.6.8 on page 120*
SMUAD	{Rd,} Rn, Rm	Signed dual multiply add	Q	*3.6.9 on page 121*
SMULBB, SMULBT SMULTB, SMULTT	{Rd,} Rn, Rm	Signed multiply (halfwords)	—	*3.6.10 on page 122*
SMULL	RdLo, RdHi, Rn, Rm	Signed multiply (32 x 32), 64-bit result	—	*3.6.2 on page 111*
SSAT	Rd, #n, Rm {,shift #s}	Signed saturate	Q	*3.7.1 on page 126*
SSAT16	Rd, #n, Rm	Signed saturate 16	Q	*3.7.2 on page 127*
SSAX	{Rd,} Rn, Rm	Signed subtract and add with exchange	GE	*3.5.14 on page 98*
SSUB16	{Rd,} Rn, Rm	Signed subtract 16	—	*3.5.13 on page 97*
SSUB8	{Rd,} Rn, Rm	Signed subtract 8	—	*3.5.13 on page 97*
STM	Rn{!}, reglist	Store multiple registers, increment after	—	*3.4.6 on page 76*
STMDB, STMEA	Rn{!}, reglist	Store multiple registers, decrement before	—	*3.4.6 on page 76*
STMFD, STMIA	Rn{!}, reglist	Store multiple registers, increment after	—	*3.4.6 on page 76*
STR	Rt, [Rn, #offset]	Store register word	—	*3.4 on page 69*
STRB, STRBT	Rt, [Rn, #offset]	Store register byte	—	*3.4 on page 69*

表21. Cortex-M4指令（续）

Mnemonic	Operands	Brief description	Flags	Page
STRD	Rt, Rt2, [Rn, #offset]	Store register two words	—	*3.4.2 on page 71*
STREX	Rd, Rt, [Rn, #offset]	Store register exclusive	—	*3.4.8 on page 79*
STREXB	Rd, Rt, [Rn]	Store register exclusive byte	—	*3.4.8 on page 79*
STREXH	Rd, Rt, [Rn]	Store register exclusive halfword	—	*3.4.8 on page 79*
STRH, STRHT	Rt, [Rn, #offset]	Store register halfword	—	*3.4 on page 69*
STRT	Rt, [Rn, #offset]	Store register word	—	*3.4 on page 69*
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V	*3.5.1 on page 83*
SUB, SUBW	{Rd,} Rn, #imm12	Subtract	N,Z,C,V	*3.5.1 on page 83*
SVC	#imm	Supervisor call	—	*3.11.10 on page 190*
SXTAB	{Rd,} Rn, Rm,{,ROR #}	Extend 8 bits to 32 and add	—	*3.8.3 on page 137*
SXTAB16	{Rd,} Rn, Rm,{,ROR #}	Dual extend 8 bits to 16 and add	—	*3.8.3 on page 137*
SXTAH	{Rd,} Rn, Rm,{,ROR #}	Extend 16 bits to 32 and add	—	*3.8.3 on page 137*
SXTB16	{Rd,} Rm {,ROR #n}	Signed extend byte 16	—	*3.8.2 on page 136*
SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	—	*3.9.3 on page 141*
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a halfword	—	*3.9.3 on page 141*
TBB	[Rn, Rm]	Table branch byte	—	*3.9.8 on page 147*
TBH	[Rn, Rm, LSL #1]	Table branch halfword	—	*3.9.8 on page 147*
TEQ	Rn, Op2	Test equivalence	N,Z,C	*3.5.9 on page 93*
TST	Rn, Op2	Test	N,Z,C	*3.5.9 on page 93*
UADD16	{Rd,} Rn, Rm	Unsigned add 16	GE	*3.5.16 on page 100*
UADD8	{Rd,} Rn, Rm	Unsigned add 8	GE	*3.5.16 on page 100*
USAX	{Rd,} Rn, Rm	Unsigned subtract and add with exchange	GE	*3.5.17 on page 101*
UHADD16	{Rd,} Rn, Rm	Unsigned halving add 16	—	*3.5.18 on page 102*
UHADD8	{Rd,} Rn, Rm	Unsigned halving add 8	—	*3.5.18 on page 102*
UHASX	{Rd,} Rn, Rm	Unsigned halving add and subtract with exchange	—	*3.5.19 on page 103*
UHSAX	{Rd,} Rn, Rm	Unsigned halving subtract and add with exchange	—	*3.5.19 on page 103*
UHSUB16	{Rd,} Rn, Rm	Unsigned halving subtract 16	—	*3.5.20 on page 104*
UHSUB8	{Rd,} Rn, Rm	Unsigned halving subtract 8	—	*3.5.20 on page 104*
UBFX	Rd, Rn, #lsb, #width	Unsigned bit field extract	—	*3.9.2 on page 140*

表21. Cortex-M4指令（续）

Mnemonic	Operands	Brief description	Flags	Page
UDIV	{Rd,} Rn, Rm	Unsigned divide	—	*3.6.3 on page 112*
UMAAL	RdLo, RdHi, Rn, Rm	Unsigned multiply accumulate accumulate long (32 x 32 + 32 +32), 64-bit result	—	*3.6.2 on page 111*
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned multiply with accumulate (32 x 32 + 64), 64-bit result	—	*3.6.2 on page 111*
UMULL	RdLo, RdHi, Rn, Rm	Unsigned multiply (32 x 32), 64-bit result	—	*3.6.2 on page 111*
UQADD16	{Rd,} Rn, Rm	Unsigned saturating add 16	—	*3.7.7 on page 132*
UQADD8	{Rd,} Rn, Rm	Unsigned saturating add 8	—	*3.7.7 on page 132*
UQASX	{Rd,} Rn, Rm	Unsigned saturating add and subtract with exchange	—	*3.7.6 on page 131*
UQSAX	{Rd,} Rn, Rm	Unsigned saturating subtract and add with exchange	—	*3.7.6 on page 131*
UQSUB16	{Rd,} Rn, Rm	Unsigned saturating subtract 16	—	*3.7.7 on page 132*
UQSUB8	{Rd,} Rn, Rm	Unsigned saturating subtract 8	—	*3.7.7 on page 132*
USAD8	{Rd,} Rn, Rm	Unsigned sum of absolute differences	—	*3.5.22 on page 106*
USADA8	{Rd,} Rn, Rm, Ra	Unsigned sum of absolute differences and accumulate	—	*3.5.23 on page 107*
USAT	Rd, #n, Rm {,shift #s}	Unsigned saturate	Q	*3.7.1 on page 126*
USAT16	Rd, #n, Rm	Unsigned saturate 16	Q	*3.7.2 on page 127*
UASX	{Rd,} Rn, Rm	Unsigned add and subtract with exchange	GE	*3.5.17 on page 101*
USUB16	{Rd,} Rn, Rm	Unsigned subtract 16	GE	*3.5.24 on page 108*
USUB8	{Rd,} Rn, Rm	Unsigned subtract 8	GE	*3.5.24 on page 108*
UXTAB	{Rd,} Rn, Rm,{,ROR #}	Rotate, extend 8 bits to 32 and add	—	*3.8.3 on page 137*
UXTAB16	{Rd,} Rn, Rm,{,ROR #}	Rotate, dual extend 8 bits to 16 and add	—	*3.8.3 on page 137*
UXTAH	{Rd,} Rn, Rm,{,ROR #}	Rotate, unsigned extend and add halfword	—	*3.8.3 on page 137*
UXTB	{Rd,} Rm {,ROR #n}	Zero extend a byte	—	*3.8.2 on page 136*
UXTB16	{Rd,} Rm {,ROR #n}	Unsigned extend byte 16	—	*3.8.2 on page 136*
UXTH	{Rd,} Rm {,ROR #n}	Zero extend a halfword	—	*3.8.2 on page 136*
VABS.F32	Sd, Sm	Floating-point absolute	—	*3.10.1 on page 151*
VADD.F32	{Sd,} Sn, Sm	Floating-point add	—	*3.10.2 on page 152*

表21. Cortex-M4指令（续）

Mnemonic	Operands	Brief description	Flags	Page
VCMP.F32	Sd, <Sm | #0.0>	Compare two floating-point registers, or one floating-point register and zero	FPSCR	*3.10.3 on page 153*
VCMPE.F32	Sd, <Sm | #0.0>	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	FPSCR	*3.10.3 on page 153*
VCVT.S32.F32	Sd, Sm	Convert between floating-point and integer	—	*3.10.4 on page 154*
VCVT.S16.F32	Sd, Sd, #fbits	Convert between floating-point and fixed point	—	*3.10.4 on page 154*
VCVTR.S32.F32	Sd, Sm	Convert between floating-point and integer with rounding	—	*3.10.4 on page 154*
VCVT<B|H>.F32.F16	Sd, Sm	Converts half-precision value to single-precision	—	*3.10.5 on page 155*
VCVTT<B|T>.F32.F16	Sd, Sm	Converts single-precision register to half-precision	—	*3.10.6 on page 156*
VDIV.F32	{Sd,} Sn, Sm	Floating-point divide	—	*3.10.7 on page 157*
VFMA.F32	{Sd,} Sn, Sm	Floating-point fused multiply accumulate	—	*3.10.8 on page 158*
VFNMA.F32	{Sd,} Sn, Sm	Floating-point fused negate multiply accumulate	—	*3.10.9 on page 159*
VFMS.F32	{Sd,} Sn, Sm	Floating-point fused multiply subtract	—	*3.10.8 on page 158*
VFNMS.F32	{Sd,} Sn, Sm	Floating-point fused negate multiply subtract	—	*3.10.9 on page 159*
VLDM.F<32|64>	Rn{!}, list	Load multiple extension registers	—	*3.10.10 on page 160*
VLDR.F<32|64>	<Dd|Sd>, [Rn]	Load an extension register from memory	—	*3.10.11 on page 161*
VLMA.F32	{Sd,} Sn, Sm	Floating-point multiply accumulate	—	*3.10.12 on page 162*
VLMS.F32	{Sd,} Sn, Sm	Floating-point multiply subtract	—	*3.10.12 on page 162*
VMOV.F32	Sd, #imm	Floating-point move immediate	—	*3.10.13 on page 163*
VMOV	Sd, Sm	Floating-point move register	—	*3.10.14 on page 164*
VMOV	Sn, Rt	Copy Arm core register to single precision	—	*3.10.18 on page 168*
VMOV	Sm, Sm1, Rt, Rt2	Copy 2 Arm core registers to 2 single precision	—	*3.10.17 on page 167*

表21. Cortex-M4指令（续）

Mnemonic	Operands	Brief description	Flags	Page
VMOV	Dd[x], Rt	Copy Arm core register to scalar	—	*3.10.15 on page 165*
VMOV	Rt, Dn[x]	Copy scalar to Arm core register	—	*3.10.16 on page 166*
VMRS	Rt, FPSCR	Move FPSCR to Arm core register or APSR	N,Z,C,V	*3.10.19 on page 169*
VMSR	FPSCR, Rt	Move to FPSCR from Arm Core register	FPSCR	*3.10.20 on page 170*
VMUL.F32	{Sd,} Sn, Sm	Floating-point multiply	—	*3.10.21 on page 171*
VNEG.F32	Sd, Sm	Floating-point negate	—	*3.10.22 on page 172*
VNMLA.F32	Sd, Sn, Sm	Floating-point multiply and add	—	*3.10.23 on page 173*
VNMLS.F32	Sd, Sn, Sm	Floating-point multiply and subtract	—	*3.10.23 on page 173*
VNMUL	{Sd,} Sn, Sm	Floating-point multiply	—	*3.10.23 on page 173*
VPOP	list	Pop extension registers	—	*3.10.24 on page 174*
VPUSH	list	Push extension registers	—	*3.10.25 on page 175*
VSQRT.F32	Sd, Sm	Calculates floating-point square root	—	*3.10.26 on page 176*
VSTM	Rn{!}, list	Floating-point register store multiple	—	*3.10.27 on page 177*
WFE	—	Wait for event	—	*3.11.11 on page 191*
WFI	—	Wait for interrupt	—	*3.11.12 on page 192*

3.2 CMSIS内联函数

ISO/IEC C代码无法直接访问某些Cortex-M4指令。本节描述了由CMIS提供的、能够生成这些指令的内联函数，以及可能由C编译器提供的函数。如果C编译器不支持相应的内联函数，您可能需要使用内联汇编器来访问某些指令。

CMSIS 提供了在 *Table 22* 中列出的内联函数，以生成 ANSI 无法直接访问的指令。

IS内联函数用于生成某些Cortex-M4在 说明

Instruction	CMSIS intrinsic function
CPSIE I	void __enable_irq(void)
CPSID I	void __disable_irq(void)
CPSIE F	void __enable_fault_irq(void)
CPSID F	void __disable_fault_irq(void)
ISB	void __ISB(void)
DSB	void __DSB(void)
DMB	void __DMB(void)
REV	uint32_t __REV(uint32_t int value)
REV16	uint32_t __REV16(uint32_t int value)
REVSH	uint32_t __REVSH(uint32_t int value)
RBIT	uint32_t __RBIT(uint32_t int value)
SEV	void __SEV(void)
WFE	void __WFE(void)
WFI	void __WFI(void)

CMSIS 还提供了若干功能，用于通过 MRS 和 MSR 指令访问特殊寄存器（参见 *Table 23*）。

表格22. CMS表23 . CMSIS内联函数用于访问特殊寄存器 客人 {v*}

Special register	Access	CMSIS function
PRIMASK	Read	uint32_t __get_PRIMASK (void)
	Write	void __set_PRIMASK (uint32_t value)
FAULTMASK	Read	uint32_t __get_FAULTMASK (void)
	Write	void __set_FAULTMASK (uint32_t value)
BASEPRI	Read	uint32_t __get_BASEPRI (void)
	Write	void __set_BASEPRI (uint32_t value)
CONTROL	Read	uint32_t __get_CONTROL (void)
	Write	void __set_CONTROL (uint32_t value)
MSP	Read	uint32_t __get_MSP (void)
	Write	void __set_MSP (uint32_t TopOfMainStack)
PSP	Read	uint32_t __get_PSP (void)
	Write	void __set_PSP (uint32_t TopOfProcStack)

3.3 关于指令描述 {v*} ONS

以下部分提供了关于使用指令的更多信息:

- *Operands on page 60*
- *Restrictions when using PC or SP on page 60*
- *Flexible second operand on page 60*
- *Shift operations on page 62*
- *Address alignment on page 65*
- *PC-relative expressions on page 65*
- *Conditional execution on page 65*
- *Instruction width selection on page 68*

3.3.1 操作数

指令操作数可以是Arm寄存器、常量或另一个指令特定参数。指令对操作数进行操作，并且通常将结果存储在目标寄存器中。当指令中存在目标寄存器时，它通常在操作数之前指定。

某些指令中的操作数是灵活的，因为它们可以是寄存器或常量（参见 *Flexible second operand*）。

3.3.2 使用 PC 或 SP 时的限制

许多指令对操作数或目标寄存器是否可以使用 *program counter* (PC) 或 *stack pointer* (SP) 有限制。如需更多信息，请参阅指令描述。

任何写入PC的地址的位[0]在使用BX、BLX、LDM、LDR或POP指令时必须为1以确保正确执行，因为该位指示所需的指令集，而Cortex-M4处理器仅支持Thumb指令。

3.3.3 可变的第二个操作数

许多通用数据处理指令具有一个灵活的第二个操作数。这在每个指令的语法描述中表示为 *operand2*。

Operand2 可以是:

- *Constant*
- *Register with optional shift*

常数{v*}

您以 *#constant,* where *constant* 的形式指定操作数2常量，其中 *constant* 可以是：

- 任何可以通过在32位字中将一个8位值左移任意位数产生的常量。

- 任何常量形式为0x00XY00XY
- 任何常量的形式为 0xXY00XY00
- 任何形式为 0xXYXYXYXY 的常量

如上所示的常量中，X 和 Y 是十六进制数字。

此外，在少量指令中，*constant* 可以包含更广泛的值范围。这些将在各条指令的描述中详细说明。

当使用操作数operand2常量与指令MOVS、MVNS、ANDS、ORRS、ORNS、EORS、BICS、TEQ或TST时，如果该常量大于255且可以通过移位一个8位值生成，进位标志将被更新为该常量的bit[31]位。如果操作数operand2是任何其他常量，这些指令不会影响进位标志。

指令替换

汇编器如果指定了一个非法常数，可能会生成等效指令。例如，指令 CMP Rd, #0xFFFFFFFE 可能被汇编为 指令 CMN Rd, #0x2 的等效形式。

带可选移位的寄存器

操作数2寄存器以 *Rm* {, *shift*} 的形式指定，其中：

- *Rm*是保存第二个操作数数据的寄存器
- *Shift*是可选的移位操作，应用于 *Rm*。它可以是以下之一：ASR #*n:* 算术右移 *n* 位，$1 \le n \le 32$ LSL #*n:* 逻辑左移 *n* 位，$1 \le n \le 31$ LSR #*n:* 逻辑右移 *n* 位，$1 \le n \le 32$ ROR #*n:* 右移 *n* 位，$1 \le n \le 31$ RRX：右移一位并扩展 —：若省略则不执行移位，等同于 LSL #0

如果您省略移位，或指定LSL #0，指令将使用*Rm*中的值。

如果您指定了移位操作，移位操作将应用于 *Rm* 中的值，并生成的 32 位值为被指令使用。然而，*Rm* 寄存器中的内容保持不变。在与某些指令一起使用时，指定带移位的寄存器也会更新进位标志。有关移位操作及其如何影响进位标志的信息，请参见 *Shift operations*。

3.3.4 移位操作

寄存器移位操作将寄存器中的位向左或向右移动指定的位数，即*shift length*。寄存器移位可以执行：

- 直接通过指令ASR、LSR、LSL、ROR和RRX。结果被写入目标寄存器。
- 在操作数2的计算过程中，由指定第二个操作数为带移位的寄存器的指令（参见 *Flexible second operand on page 60*）执行。结果被该指令使用。

允许的移位长度取决于移位类型和指令（请参阅各指令的描述或 *Flexible second operand*）。如果移位长度为 0，则不执行移位。寄存器移位操作会更新进位标志，除非指定的移位长度为 0。以下子节描述了各种移位操作及其对进位标志的影响。在这些描述中，*Rm* 是包含待移位值的寄存器，*n* 是移位长度。

自动语音识别

算术右移 *n* 位将 *Rm* 寄存器左边的 32-*n* 位向右移动 *n* 位，进入结果的右边 32-*n* 位中。并将寄存器的原始 bit[31] 复制到结果的左边 *n* 位中（参见 *Figure 13: ASR #3 on page 62*）。

您可以使用ASR *#n*操作将*Rm* 寄存器中的值除以2^n，并将结果向负无穷取整。

当指令为ASRS，或在操作数2中使用ASR #n与以下指令时：MOVS、MVNS、ANDS、ORRS、ORNS、EORS、BICS、TEQ或TST，进位标志会被更新为*Rm* 寄存器的bit[*n*-1]位。

Note: 1 If n is 32 or more, all the bits in the result are set to the value of bit[31] of Rm.

2 If n is 32 or more and the carry flag is updated, it is updated to the value of bit[31] of Rm.

图13. ASR #3

LSR

逻辑右移 *n* 位将 *Rm* 寄存器左侧的 32-*n* 位向右移动 *n* 位，进入结果的右侧 32-*n* 位。并将结果的左侧 *n* 位设置为 0（参见 *Figure 14*）。

您可以使用LSR #n操作将*Rm* 寄存器中的值除以2^n，如果该值被视为无符号整数。

当指令为LSRS，或在*operand2*中使用LSR #n与指令MOVS、MVNS、ANDS、ORRS、ORNS、EORS、BICS、TEQ或TST时，进位标志会被更新为最后移出的位bit[*n*-1]，即*Rm* 寄存器中的位。

Note: 1 If n is 32 or more, then all the bits in the result are cleared to 0.

2 If n is 33 or more and the carry flag is updated, it is updated to 0.

图14. LSR #3

林登脚本语言

逻辑左移 *n* 位将 *Rm* 寄存器的右侧 32-*n* 位向左移动 *n* 位，进入结果的左侧 32-*n* 位。并将结果的右侧 *n* 位设置为 0（参见 *Figure 15: LSL #3*）。

LSL #n 操作可以用于将 *Rm* 寄存器中的值乘以 2^n，如果该值被视为无符号整数或二进制补码有符号整数。溢出可能在没有预警的情况下发生。

当指令为LSLS，或当使用LSL *#n*且*n*非零时，在*operand2*中使用MOVS、MVNS、ANDS、ORRS、ORNS、EORS、BICS、TEQ或TST指令，进位标志会被更新为从*Rm* 寄存器中最后移出的位，即bit[32-*n*]。这些指令在与LSL #0一起使用时不会影响进位标志。

Note: 1 If n is 32 or more, then all the bits in the result are cleared to 0.

2 If n is 33 or more and the carry flag is updated, it is updated to 0.

图15. LSL #3

收入回报率

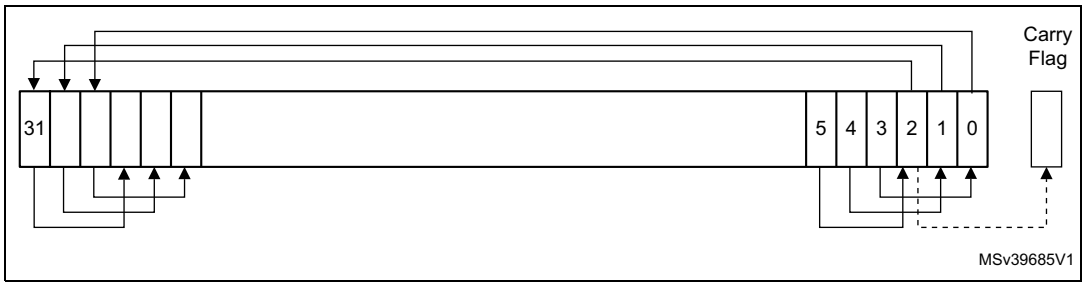
向右旋转 *n* 位将 *Rm* 寄存器的左边 32-*n* 位向右移动 *n* 位，移入结果的右边 32-*n* 位。它还将寄存器的右边 *n* 位移动到结果的左边 *n* 位（参见 *Figure 16*）。

当指令为RORS，或在*operand2*中与MOVS、MVNS、ANDS、ORRS、ORNS、EORS、BICS、TEQ或TST指令一起使用ROR #*n*时，进位标志会被更新为*Rm*寄存器的最后一位旋转结果，即bit[*n*-1]。

Note: 1 If n is 32, then the value of the result is same as the value in Rm, and if the carry flag is updated, it is updated to bit[31] of Rm.

2 ROR with shift length, n, more than 32 is the same as ROR with shift length n-32.

图16. ROR #3



RRX

带扩展的右旋转将*Rm*寄存器的位向右移动一位。并将进位标志复制到结果的bit[31]位（参见*Figure 17*）。

当指令为RRXS或在operand2中使用RRX指令与MOVS、MVNS、ANDS、ORRS、ORNS、EORS、BICS、TEQ或TST指令时，进位标志会被更新为*Rm*寄存器的bit[0]。

图17. RRX #3

3.3.5 地址对齐

对齐访问是指一种操作，其中使用字对齐地址进行字、双字或多字访问，或使用半字对齐地址进行半字访问。字节访问始终对齐。

Cortex-M4处理器仅支持以下指令的非对齐访问：{v*}

- 加载寄存器, 加载寄存器并转移
- 加载寄存器半字, 加载寄存器半字带转移
- 加载寄存器带符号扩展半字指令, 加载寄存器带符号扩展半字指令
- 字符串, 字符串开始
- 战略人力资源, 战略人力资源培训

所有其他加载和存储指令如果执行未对齐的访问，将生成使用故障异常，因此它们的访问必须进行地址对齐。如需了解有关使用故障的更多信息，请参阅 *Fault handling on page 44*。

未对齐的访问通常比对齐的访问要慢。此外，某些内存区域可能不支持未对齐的访问。因此，Arm建议程序员确保访问对齐。为避免意外生成未对齐的访问，请使用配置和控制寄存器中的UNALIGN_TRP位来拦截所有未对齐的访问，参见 *Configuration and control register (CCR) on page 231*。

3.3.6 相对程序计数器的表达式

PC相对表达式或 *label* 是一个符号，表示指令或字面数据的地址。它在指令中表示为程序计数器值加上或减去一个数值偏移量。汇编器根据标签和当前指令的地址计算所需的偏移量。如果偏移量过大，汇编器将产生错误。

- 对于B、BL、CBNZ和CBZ指令，程序计数器（PC）的值是当前指令地址加上四个字节。
- 对于所有其他使用标签的指令，PC的值是当前指令的地址加上四个字节，并将结果的bit[1]清零以实现字对齐。
- 您的汇编器可能允许其他用于PC相对表达式的语法，例如标签加上或减去一个数字，或者形式为[PC, #number]的表达式。

3.3.7 条件执行

大多数数据处理指令可根据操作结果可选地更新*application program status register*（APSR）中的条件标志（见*Application program status register on page 21*）。某些指令会更新所有标志，而另一些仅更新部分标志。如果某个标志未被更新，则保留其原始值。请参阅影响标志的指令描述。

您可以有条件地执行一条指令，基于另一条指令设置的条件标志：

- 在更新标志的指令之后立即
- 在任何数量的中间指令之后，这些指令未更新标志

条件执行可以通过使用条件分支或通过在指令后添加条件码后缀来实现。有关在指令后添加的后缀以将其转换为条件指令的列表，请参见 *Table 24: Condition code suffixes on page 67* 。条件码后缀使处理器能够根据标志测试条件。如果条件指令的条件测试失败，则指令：

- 不执行
- 不将其任何值写入目标寄存器。
- 不会影响任何标志。
- 不会生成任何异常。

条件指令（除条件分支外）必须位于If-then指令块内。有关在使用IT指令时的更多信息和限制，请参见*IT on page 145*。根据供应商的不同，如果存在位于IT指令块外的条件指令，汇编器可能会自动插入IT指令。

使用 CBZ 和 CBNZ 指令来比较寄存器的值与零，并根据结果进行分支。

本节描述：

- *The condition flags*
- *Condition code suffixes on page 67*

条件标志

APSR 包含以下条件标志：

- N: 当操作结果为负数时置1，否则清0。
- Z: 当操作结果为零时设置为1，否则清零。
- C：当操作产生进位时置为1，否则清零为0。
- {v*}：当操作导致溢出时设置为1，否则清除为0。

如需了解有关APSR的更多信息，请参见 *Program status register on page 19*。

发生进位：

- 如果加法的结果大于或等于232。
- 如果减法运算的结果为非负数。
- 作为移动指令或逻辑指令中内联桶形移位器操作的结果。{v*}

溢出发生在以下情况：如果结果的符号与在无限精度下执行该操作所得到的结果的符号不一致，例如：

- 如果将两个负值相加得到正值。
- 如果两个正数相加的结果是负数。
- 如果从负数中减去正数会生成正数。
- 如果从正数中减去负数生成负数。

比较操作与减法（CMP）或加法（CMN）相同，除了结果被丢弃。如需更多信息，请参阅指令描述。

大多数指令仅在指定了S后缀时才会更新状态标志。请参阅指令描述以获取更多信息。

条件码后缀

可以有条件执行的指令具有一个可选的条件码，语法描述中显示为{*cond*}。条件执行需要一个前置的IT指令。带有条件码的指令只有在APSR中的条件码标志满足指定条件时才会执行。*Table 24* 显示要使用的条件码。

你可以通过IT指令实现条件执行，从而减少代码中的分支指令数量。

*Table 24*还展示了条件码后缀与N、Z、C和V标志之间的关系。

表24. 条件码后缀

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned ≥
CC or LO	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N != V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N != V	Less than or equal, signed ≤
AL	Can have any value	Always. This is the default when no suffix is specified.

*Specific example 1: Absolute value*展示了一个使用条件指令来求取数字的绝对值。R0 = ABS(R1)。

具体示例 1: 绝对值

```
MOV R0, R1; R0 = R1, 设置标志
IT MI ; IT 指令用于负条件 RSBMIR0, R1, #0;    如果为负,R0 = -R1
```

*Specific example 2: Compare and update value*展示了如何使用条件指令来更新R4的值，当R0和R2的带符号值分别大于R1和R3时。

具体示例2：比较并更新值

```
CMP R0, R1 ; 比较 R0 和 R1，设置标志 ITT GT ; IT 指令用于两个 GT 条件
```

CMPGT R2, R3; 如果'大于'，比较 R2 和 R3，设置标志 MOVGT R4, R5 ; 如果仍然'大于'，执行 R4 = R5

3.3.8 指令宽度选择

有许多指令可以根据指定的操作数和目标寄存器生成16位编码或32位编码。对于这些指令中的一些，您可以使用指令宽度后缀来强制指定特定的指令大小。 .W后缀强制生成32位指令编码。.N后缀强制生成16位指令编码。

如果您指定了指令宽度后缀，并且汇编器无法生成所请求宽度的指令编码，它会生成一个错误。

在某些情况下，可能需要指定.W后缀，例如当操作数是指令或字面数据的标签时，如分支指令的情况。原因在于汇编器可能不会自动生成正确大小的编码。

要使用指令宽度后缀，请将其立即放置在指令助记符和条件码之后（如有）。
Specific example 3: Instruction width selection 显示带有指令宽度后缀的指令。

具体示例3：指令宽度选择

```
BCS.W 标签;                生成32位指令，即使对于短跳转
ADDS.W R0, R0, R1;         尽管相同的;操作可以用16位指令完成，但仍生成一个32位指令
```

3.4 内存访问指令

*Table 25*显示内存访问指令: {v*}

表25. 内存访问指令

Mnemonic	Brief description	See
ADR	Load PC-relative address	*ADR on page 70*
CLREX	Clear exclusive	*CLREX on page 80*
LDM{mode}	Load multiple registers	*LDM and STM on page 76*
LDR{type}	Load register using immediate offset	*LDR and STR, immediate offset on page 71*
LDR{type}	Load register using register offset	*LDR and STR, register offset on page 73*
LDR{type}T	Load register with unprivileged access	*LDR and STR, unprivileged on page 74*
LDR	Load register using PC-relative address	*LDR, PC-relative on page 75*
LDRD	Load register dual	*LDR and STR, immediate offset on page 71*
LDREX{type}	Load register exclusive	*LDREX and STREX on page 79*
POP	Pop registers from stack	*PUSH and POP on page 78*
PUSH	Push registers onto stack	*PUSH and POP on page 78*
STM{mode}	Store multiple registers	*LDM and STM on page 76*
STR{type}	Store register using immediate offset	*LDR and STR, immediate offset on page 71*
STR{type}	Store register using register offset	*LDR and STR, register offset on page 73*
STR{type}T	Store register with unprivileged access	*LDR and STR, unprivileged on page 74*
STREX{type}	Store register exclusive	*LDREX and STREX on page 79*

3.4.1 ADR

加载PC相对地址。

语法

ADR{cond} Rd, 标签

其中:

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。
- ‘*label*’ 是一个程序计数器相对的表达式（参见 *PC-relative expressions on page 65*）

运营

ADR 通过将立即数加到程序计数器（PC）上确定地址。它将结果写入目标寄存器。

ADR 产生 位置无关代码，因为地址是PC -相对的.

如果你使用ADR生成BX或BLX指令的目标地址，必须确保生成的地址的bit[0]设置为1以确保正确执行。

label 的值必须位于以 PC 中的地址为基准的 -4095 到 4095 范围内。

Note: You might have to use the .W suffix to get the maximum offset range or to generate addresses that are not word-aligned (see Instruction width selection on page 68).

限制

*Rd*必须既不是SP也不是PC。

条件标志

该指令不会改变标志。

示例

```
ADR R1, TextMessage; 将标记为 ; TextMessage 的位置的地址值写入 R1
```

3.4.2 加载寄存器和存储寄存器，立即数偏移

加载和存储带立即偏移量、预索引立即偏移量或后索引立即偏移量。

语法

op{type}{cond} Rt, [Rn {, #offset}]; 立即偏移量 op{type}{cond} Rt, [Rn, #offset]!; 预索引 op{type}{cond} Rt, [Rn], #offset; 后索引 opD{cond} Rt, Rt2, [Rn {, #offset}]; 立即偏移量，两个字 opD{cond} Rt, Rt2, [Rn, #offset]!; 预索引，两个字 opD{cond} Rt, Rt2, [Rn], #offset; 后索引，两个字

其中：

- ‘*op*’ 是 LDR（加载寄存器）或 STR（存储寄存器）
- ‘*type*’ 是以下之一：B: 无符号字节，加载时零扩展到32位 SB: 有符号字节，符号扩展到32位（仅LDR） H: 无符号半字，加载时零扩展到32位 SH: 有符号半字，符号扩展到32位（仅LDR） —: 省略，用于字

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rt*’ 是用于加载或存储的寄存器
- ‘*Rn*’ 是内存地址所基于的寄存器
- ‘*offset*’ 是相对于 *Rn* 的偏移量。如果省略 *offset*，则地址为 *Rn* 的内容。
- ‘*Rt2*’ 是用于双字操作的附加寄存器

运营

LDR指令从内存中加载一个或两个寄存器的值。STR指令将一个或两个寄存器的值存储到内存中。

带有立即偏移量的加载和存储指令可以使用以下寻址方式：{v*}

偏移寻址 {v*}

偏移量值被加到或从寄存器 *Rn* 获取的地址上减去。结果被用作内存访问的地址。寄存器 *Rn* 保持不变。这种模式的汇编语言语法为：[*Rn*, #*offset*].

预索引寻址

偏移值被加到或从寄存器 *Rn* 获取的地址中减去。结果被用作内存访问的地址，并写回寄存器 *Rn*。此模式的汇编语言语法为：[*Rn*, #*offset*]!

后索引寻址

从寄存器 *Rn* 获取的地址被用作内存访问的地址。偏移值被加到或从该地址减去，并写回寄存器 *Rn*。该模式的汇编语言语法为：[*Rn*], #*offset*.

加载或存储的值可以是字节、半字、字或两个字。字节和半字可以是有符号或无符号的（参见 *Address alignment on page 65*）。

*Table 26*显示立即数、预索引和后索引形式的偏移量范围。

表26. 立即、前索引和后索引的偏移范围

Instruction type	Immediate offset	Pre-indexed	Post-indexed
Word, halfword, signed halfword, byte, or signed byte	-255 to 4095	-255 to 255	-255 to 255
Two words	Multiple of 4 in the range -1020 to 1020	Multiple of 4 in the range -1020 to 1020	Multiple of 4 in the range -1020 to 1020

限制

- 对于加载指令:
  – *Rt* 仅用于字加载，可为 SP 或 PC。– *Rt* 在双字加载时必须与 *Rt2* 不同。– *Rn* 在预索引或后索引形式中必须与 *Rt* 和 *Rt2* 不同。

- 当 *Rt* 是字加载指令中的 PC 时。– 加载值的 bit[0] 必须为 1 以确保正确执行。– 会跳转到通过将加载值的 bit[0] 改为 0 而生成的地址。– 如果指令是条件指令，则必须是 IT 块中的最后一条指令。

- 关于存储指令:
  – *Rt* 可以是 SP，仅用于单词存储。– *Rt* 不得为 PC。– *Rn* 不得为 PC。– *Rn* 必须与 *Rt* 和 *Rt2* 在预索引或后索引形式中不同

条件标志

这些指令不会更改标志。

示例

LDR R8, [R10] ; 从R10中的地址加载R8。 LDRNE R2, [R5, #960]!; 条件性地加载R2到R5中的地址960字节之上，并将R5增加960。 STR R2, [R9,#const-struc]; const-struc是一个用于计算的表达式，评估为0-4095范围内的常量。 STRH R3, [R4], #4; 将R3作为半字数据存储到R4中的地址，然后将R4增加4 LDRD R8, R9, [R3, #0x20]; 从R3中的地址32字节之上加载R8，以及从R3中的地址36字节之上加载R9 STRD R0, R1, [R8], #-16; 将R0存储到R8中的地址，将R1存储到R8中的地址4字节之上，并将R8减少16。

3.4.3 LDR 和 STR，寄存器偏移量

带寄存器偏移量的加载和存储

语法

操作{类型}{条件} Rt, [Rn, Rm {, LSL #n}]

其中:

- ‘*op*’ 可以是 LDR（加载寄存器）或 STR（存储寄存器）。
- "*type*" 是以下之一： B: 无符号字节，在加载时零扩展到32位。 SB: 有符号字节，符号扩展到32位（仅适用于LDR）。 H: 无符号半字，在加载时零扩展到32位。 SH: 有符号半字，符号扩展到32位（仅适用于LDR）。 —: 忽略，用于字。
- ‘*cond*’ 是一个可选的条件码，参见 *Conditional execution on page 65*。
- ‘*Rt*’ 是用于加载或存储的寄存器。
- ‘*Rn*’ 是内存地址所基于的寄存器。
- ‘*Rm*’ 是一个包含用作偏移量的值的寄存器。
- ‘*LSL #n*’ 是一个可选的偏移量，其中 *n* 的范围是 0 到 3。

运营

LDR指令将寄存器从内存加载一个值。STR指令将寄存器值存储到内存中。用于加载或存储的内存地址相对于寄存器 *Rn* 的偏移量。偏移量由 *Rm* 寄存器指定，并可通过LSL最多左移3位。加载或存储的值可以是字节、半字或字。对于加载指令，字节和半字可以是带符号或无符号的（参见 *Address alignment on page 65*）。

限制

在这些说明中:

- *Rn*不得为PC。
- *Rm*必须既不是SP也不是PC。
- *Rt*只能作为字加载和字存储的SP。
- *Rt*可以是仅限PC用于Word加载。

当 *Rt* 是字加载指令中的PC时:

- 加载值的bit[0]必须为1以确保正确执行，并且会跳转到此半字对齐的地址。
- 如果该指令是条件性的，它必须是IT块中的最后一条指令。

条件标志

这些指令不会更改标志。

示例

STR R0, [R5, R1]; 存储 R0 的值到地址等于 {v*}； R5 和 R1 的和LDRSB R0, [R5, R1, LSL #1]; 从地址等于 {v*} 的位置读取字节值，并进行符号扩展

将值转换为字并存入R0 STR R0, [R1, R2, LSL #2]; 存储R0到地址，该地址等于R1与四倍R2的和

3.4.4 加载寄存器和存储寄存器，非特权

加载和存储具有非特权访问

语法

op{type}T{cond} Rt, [Rn {, #offset}]; 立即数偏移量 其中:

- ‘*op*’ 可以是 LDR（加载寄存器）或 STR（存储寄存器）。
- ‘*type*’ 是以下之一：B: 无符号字节，加载时零扩展为32位。SB: 带符号字节，符号扩展为32位（仅限LDR）。H: 无符号半字，加载时零扩展为32位。SH: 带符号半字，符号扩展为32位（仅限LDR）。—: 省略，用于字（32位）。

- ‘*cond*’ 是一个可选的状态码，参见 *Conditional execution on page 65*。
- “*Rt*”是用于加载或存储的寄存器。
- ‘*Rn*’ 是内存地址所基于的寄存器。
- ‘*offset*’ 是 *Rn* 的一个偏移量，其范围为 0 到 255。如果省略 *offset*，则地址为 *Rn* 中的值。

运营

这些加载和存储指令执行与带有立即偏移量的内存访问指令相同的功能（参见 *LDR and STR, immediate offset on page 71*）。区别在于，这些指令即使在特权软件中使用，也仅允许非特权访问。

当在非特权软件中使用时，这些指令的行为与带有立即偏移量的常规内存访问指令完全相同。

限制

在这些说明中:

- *Rn*不得为PC。
- *Rt*必须既不是SP也不是PC。

条件标志

这些指令不会更改标志。

示例

STRBTEQ R4, [R7] ; 有条件地将R4中的最低有效字节存储到R7中的地址，非特权访问 LDRHT R2, [R2, #8]; 从等于R2和8之和的地址加载半字值到R2，非特权访问

3.4.5 LDR，PC相对的

从内存加载寄存器

语法

LDR{type}{cond} Rt, label LDRD{cond} Rt, Rt2, label; 加载两个字

其中:

- ‘*type*’ 是以下之一：B: 无符号字节，零扩展为32位。 SB: 带符号字节，符号扩展为32位。 H: 无符号半字，符号扩展为32位。 SH: 带符号半字，符号扩展为32位。 —: 省略，用于字。
- ‘*cond*’ 是一个可选的条件码，参见 *Conditional execution on page 65*。
- ‘*Rt*’ 是用于加载或存储的寄存器。
- ‘*Rt2*’ 是用于加载或存储的第二寄存器。
- ‘*label*’ 是一个 PC相对表达式，参见 *PC-relative expressions on page 65*。’

运营

LDR 从 PC 相对的内存地址加载一个寄存器中的值。 内存地址由标签或 PC 的偏移量指定。要加载或存储的值可以是字节、半字或字。对于加载指令，字节和半字可以是有符号的或无符号的（参见 *Address alignment on page 65*）。

*‘label’*必须位于当前指令的有限范围内。*Table 27* 显示 *label* 与程序计数器之间的可能偏移量。您可能需要使用 .W 后缀以获得最大偏移量范围（参见 *Instruction width selection on page 68*）。

表 27. ***label***-PC 偏移范围

Instruction type	Offset range
Word, halfword, signed halfword, byte, signed byte	–4095 to 4095
Two words	–1020 to 1020

限制

在这些说明中:

- *Rt2*必须既不是SP也不是PC
- *Rt*必须与*Rt2*不同
- *Rt*可以是SP或PC，仅用于字加载
- 当*Rt*是字加载指令中的PC时：加载值的位[0]必须为1以确保正确执行，并且会跳转到该半字对齐的地址。如果该指令是条件性的，则必须是IT块中的最后一条指令。

条件标志

这些指令不会更改标志。

示例

LDR R0, LookUpTable; 从标记为 LookUpTable 的地址加载一个数据字到 R0 LDRSB R7, l
ocaldata; 从标记为 localdata 的地址加载一个字节值到 R7，进行符号扩展并将其存储为字
值

3.4.6 加载多个和存储多个

加载和存储多个寄存器。

语法

操作{addr_mode}{cond} 寄存器 Rn{!}, 寄存器列表

其中：

- ‘*op*’ 为 LDM（加载多个寄存器）或 STM（存储多个寄存器）。
- ‘*addr_mode*’ 可以是以下任何一种：IA：每次访问后递增地址（这是默认设置）。DB：每次访问前递减地址。
- ‘*cond*’ 是一个可选的状态码，参见 *Conditional execution on page 65*。
- ‘*Rn*’ 是内存地址所基于的寄存器。
- ‘*!*’ 是一个可选的写回后缀。如果存在 !，则最终地址（从其加载或存储到的地址）被写回至 *Rn*。
- ‘*reglist*’ 是一个或多个要加载或存储的寄存器列表，用大括号括起来。它可以包含寄存器范围。如果包含多个寄存器或寄存器范围，必须用逗号分隔，参见 *Examples on page 77*。

LDM 和 LDMFD 是 LDMIA 的同义指令。LDMFD 指的是其用于从满递减堆栈中弹出数据的用法。

LDMEA 是 LDMDB 的同义词，指的是其用于从空的升序栈中弹出数据。

STM 和 STMEA 是 STMIA 的同义词。STMEA 指的是将其用于将数据压入空的递增栈。

STMFD 是 STMDB 的同义词，指的是其用于将数据压入满递减堆栈的用法。

运营

LDM指令从基于*Rn*的内存地址加载字值到寄存器*reglist*中。

STM指令将寄存器*reglist*中的字值存储到基于*Rn*的内存地址。

对于LDM、LDMIA、LDMFD、STM、STMIA和STMEA，其用于访问的内存地址以4字节间隔分布在*Rn*到*Rn* + 4 * (*n*-1)的范围内，其中*n*是*reglist*中的寄存器数量。访问按寄存器编号递增的顺序进行，其中

使用最低内存地址的编号最低的寄存器和使用最高内存地址的编号最高的寄存器。如果指定了写回后缀，则值 *Rn* + 4 * (*n*-1) 会被写回至 *Rn*。

对于LDMDB、LDMEA、STMDB和STMFD，用于访问的内存地址以4字节间隔分布，范围从*Rn*到*Rn* - 4 * (*n*-1)，其中*n*是*reglist*中的寄存器数量。访问按寄存器编号递减的顺序进行，编号最高的寄存器使用最高的内存地址，编号最低的寄存器使用最低的内存地址。如果指定了写回后缀，则将值*Rn* - 4 * (*n*)写回至*Rn*。

PUSH和POP指令可以用这种形式表示（详见*PUSH and POP*）。

限制

在这些说明中:

- *Rn*不得为PC。
- *reglist*不得包含 SP。
- 在任何STM指令中，*reglist*不得包含PC。
- 在任何LDM指令中，如果包含LR，则*reglist*不得包含PC。
- *reglist*如果指定写回后缀，则不得包含 *Rn*。

当程序计数器处于LDM指令的*reglist*时:

- 加载到PC中的值的bit[0]位必须为1以确保正确执行，将发生跳转到此半字对齐的地址。
- 如果该指令是条件性的，它必须是IT块中的最后一条指令。

条件标志

这些指令不会更改标志。

示例

```
LDM R8,{R0,R2,R9} ; LDMIA 是 LDM 的同义词 STMDB R1!,{R3-R
6,R11,R12}
```

错误示例

```
STM R5!,{R5,R4,R9} ; R5存储的值不可预测  LDM R2, {} ; 列表中必须至少有一个寄存器
```

3.4.7 压栈和弹栈

将寄存器压入并从满递减堆栈弹出，其中 {v*} 表示... PUSH 和 POP 是 STMDB 和 LDM（或 LDMIA）的同义词，其访问的内存地址基于 SP，并将最终访问地址写回 SP。在这些情况下，PUSH 和 POP 是首选的助记符。

语法

压栈{cond} 寄存器列表
弹栈{cond} 寄存器列表

其中:

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。”
- ‘*reglist*’ 是一个非空的寄存器列表（或寄存器范围），用大括号括起来。必须用逗号分隔寄存器列表或范围（参见 *Examples on page 77*）。

运营

- PUSH 按寄存器编号递减的顺序将寄存器压入堆栈，最高编号的寄存器使用最高的内存地址，最低编号的寄存器使用最低的内存地址。
- POP 从堆栈中加载寄存器，按照寄存器编号递增的顺序，编号最低的寄存器使用最低的内存地址，编号最高的寄存器使用最高的内存地址。
- PUSH 使用 SP 寄存器的值减四作为最高内存地址，POP 使用 SP 寄存器的值作为最低内存地址，实现一个满减栈。完成时，PUSH 将 SP 寄存器更新为指向最低存储值的位置，而 POP 将 SP 寄存器更新为指向最高加载位置上方的位置。
- 如果POP指令在其寄存器列表中包含PC，则在POP指令完成时会执行到该位置的分支。PC读取值的Bit[0]用于更新APSR的T位。该位必须为1以确保正确操作。更多信息请参见 *LDM and STM on page 76*。

限制

在这些说明中:

- *‘reglist’*不得包含 SP。
- 对于PUSH指令，*reglist*不得包含PC。
- 对于POP指令，如果*reglist*包含LR，则不得包含PC。当PC位于*reglist*中时，在POP指令中加载到PC的值的bit[0]必须为1以确保正确执行，并将发生到该半字对齐地址的分支。如果该指令是条件指令，则必须是IT块中的最后一条指令。

条件标志

这些指令不会更改标志。

示例

PUSH {R0,R4-R7} ; 将 R0,R4,R5,R6,R7 压入堆栈 PUSH {R2,LR} ; 将 R2 和链接寄存器压入堆栈 POP {R0,R6,PC} ; 从堆栈弹出 r0,r6 和 PC，然后跳转到新的 PC.

3.4.8 加载-独占 和 存储-释放

加载和存储寄存器独占

语法

LDREX{cond} Rt, [Rn {, #offset}] STREX{cond} R
d, Rt, [Rn {, #offset}] LDREXB{cond} Rt, [Rn] STR
EXB{cond} Rd, Rt, [Rn] LDREXH{cond} Rt, [Rn] S
TREXH{cond} Rd, Rt, [Rn]

其中:

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。”
- ‘*Rd*’ 是返回状态的目标寄存器。
- ‘*Rt*’ 是用于加载或存储的寄存器。
- “*Rn*”是内存地址所基于的寄存器。
- ‘*offset*’ 是应用于 *Rn* 中值的可选偏移量。如果省略 *offset*，地址为 *Rn* 中的值。

运营

LDREX、LDREXB 和 LDREXH 分别从内存地址加载一个字、字节和半字。

STREX、STREXB 和 STREXH 分别尝试将一个字、字节和半字存储到内存地址。任何存储独占指令使用的地址必须与最近一次执行的加载独占指令的地址相同。存储独占指令存储的值也必须与前一次加载独占指令加载的值具有相同的数据大小。这意味着软件必须始终使用一个加载独占指令和一个匹配的存储独占指令来执行同步操作，参见 *Synchronization primitives on page 34*。

如果一个独占存储指令执行存储操作，它会向其目标寄存器写入0。 如果它不执行存储操作，它会向其目标寄存器写入1。 如果存储独占指令向目标寄存器写入0，它保证没有

系统中的其他进程已访问内存位置加载独占{v*}
并存储独占指令。

出于性能考虑，应将对应的加载独占和存储独占指令之间的指令数量保持在最低限度。

Note: The result of executing a store-exclusive instruction to an address that is different from that used in the preceding load-exclusive instruction is unpredictable.

限制

在这些说明中:

- 不要使用PC。
- 不要对 *Rd* 和 *Rt.* 使用 SP
- 对于STREX，*Rd*必须与*Rt*以及*Rn.*不同
- 偏移量的值必须是4的倍数，在0-1020范围内。

条件标志

这些指令不会更改标志。

示例

```
MOV R1, #0x1 ; 初始化‘lock taken’值 try LDREX R0, [LockAddr] ; 加载锁值 CMP R0, #0 ; 锁是否可用？ ITT EQ ; 用于 STREXEQ 和 CMPEQ 的 IT 指令 STREXEQ R0, R1, [LockAddr] ; 尝试获取锁 CMPEQ R0, #0 ; 此操作是否成功？ BNE try ; 不 – 再次尝试 ; 是 – 我们已获取锁
```

3.4.9 CLREX

清晰独家

语法

CLREX{cond}

其中:

‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）

运营

使用CLREX使后续的STREX、STREXB或STREXH指令向其目标寄存器写入1，并使存储操作失败。这在异常处理程序代码中很有用，当同步操作中独占加载指令与对应的独占存储指令之间发生异常时，可强制使独占存储操作失败。

参见 *Synchronization primitives on page 34* 以获取更多信息。

条件标志

这些指令不会更改标志。

示例

```
CLREX
```

3.5 通用数据处理说明

*Table 28*显示数据处理说明

表28. 数据处理说明

Mnemonic	Brief description	See
ADC	Add with carry	*ADD, ADC, SUB, SBC, and RSB on page 83*
ADD	Add	*ADD, ADC, SUB, SBC, and RSB on page 83*
ADDW	Add	*ADD, ADC, SUB, SBC, and RSB on page 83*
AND	Logical AND	*AND, ORR, EOR, BIC, and ORN on page 85*
ASR	Arithmetic Shift Right	*ASR, LSL, LSR, ROR, and RRX on page 86*
BIC	Bit Clear	*AND, ORR, EOR, BIC, and ORN on page 85*
CLZ	Count leading zeros	*CLZ on page 87*
CMN	Compare Negative	*CMP and CMN on page 88*
CMP	Compare	*CMP and CMN on page 88*
EOR	Exclusive OR	*AND, ORR, EOR, BIC, and ORN on page 85*
LSL	Logical Shift Left	*ASR, LSL, LSR, ROR, and RRX on page 86*
LSR	Logical Shift Right	*ASR, LSL, LSR, ROR, and RRX on page 86*
MOV	Move	*MOV and MVN on page 89*
MOVT	Move Top	*MOVT on page 91*
MOVW	Move 16-bit constant	*MOV and MVN on page 89*
MVN	Move NOT	*MOV and MVN on page 89*
ORN	Logical OR NOT	*AND, ORR, EOR, BIC, and ORN on page 85*
ORR	Logical OR	*AND, ORR, EOR, BIC, and ORN on page 85*
RBIT	Reverse Bits	*REV, REV16, REVSH, and RBIT on page 92*
REV	Reverse byte order in a word	*REV, REV16, REVSH, and RBIT on page 92*
REV16	Reverse byte order in each halfword	*REV, REV16, REVSH, and RBIT on page 92*
REVSH	Reverse byte order in bottom halfword and sign extend	*REV, REV16, REVSH, and RBIT on page 92*
ROR	Rotate Right	*ASR, LSL, LSR, ROR, and RRX on page 86*
RRX	Rotate Right with Extend	*ASR, LSL, LSR, ROR, and RRX on page 86*
RSB	Reverse Subtract	*ADD, ADC, SUB, SBC, and RSB on page 83*
SADD16	Signed Add 16	*SADD16 and SADD8 on page 93*
SADD8	Signed Add 8	*SADD16 and SADD8 on page 93*
SASX	Signed Add and Subtract with Exchange	*SASX and SSAX on page 98*
SSAX	Signed Subtract and Add with Exchange	*SASX and SSAX on page 98*
SBC	Subtract with Carry	*ADD, ADC, SUB, SBC, and RSB on page 83*
SHADD16	Signed Halving Add 16	*SHADD16 and SHADD8 on page 94*
SHADD8	Signed Halving Add 8	*SHADD16 and SHADD8 on page 94*

表28. 数据处理指令（续）

Mnemonic	Brief description	See
SHASX	Signed Halving Add and Subtract with Exchange	*SHASX and SHSAX on page 95*
SHSAX	Signed Halving Subtract and Add with exchange	*SHASX and SHSAX on page 95*
SHSUB16	Signed Halving Subtract 16	*SHSUB16 and SHSUB8 on page 96*
SHSUB8	Signed Halving Subtract 8	*SHSUB16 and SHSUB8 on page 96*
SSUB16	Signed Subtract 16	*SSUB16 and SSUB8 on page 97*
SSUB8	Signed subtract 8	*SSUB16 and SSUB8 on page 97*
SUB	Subtract	*ADD, ADC, SUB, SBC, and RSB on page 83*
SUBW	Subtract	*ADD, ADC, SUB, SBC, and RSB on page 83*
TEQ	Test Equivalence	*SADD16 and SADD8 on page 93*
TST	Test	*SADD16 and SADD8 on page 93*
UADD16	Unsigned Add 16	*UADD16 and UADD8 on page 100*
UADD8	Unsigned Add 8	*UADD16 and UADD8 on page 100*
UASX	Unsigned Add and Subtract with Exchange	*UASX and USAX on page 101*
USAX	Unsigned Subtract and Add with Exchange	*UASX and USAX on page 101*
UHADD16	Unsigned Halving Add 16	*UHADD16 and UHADD8 on page 102*
UHADD8	Unsigned Halving Add 8	*UHADD16 and UHADD8 on page 102*
UHASX	Unsigned Halving Add and Subtract with Exchange	*UHASX and UHSAX on page 103*
UHSAX	Unsigned Halving Subtract and Add with Exchange	*UHASX and UHSAX on page 103*
UHSUB16	Unsigned Halving Subtract 16	*UHSUB16 and UHSUB8 on page 104*
UHSUB8	Unsigned Halving Subtract 8	*UHSUB16 and UHSUB8 on page 104*
USAD8	Unsigned Sum of Absolute Differences	*USAD8 on page 106*
USADA8	Unsigned Sum of Absolute Differences and accumulate	*USADA8 on page 107*
USUB16	Unsigned Subtract 16	*USUB16 and USUB8 on page 108*
USUB8	Unsigned Subtract 8	*USUB16 and USUB8 on page 108*

3.5.1 加、带进位加、减、带进位减以及反向减

加法、带进位加法、减法、带进位减法、以及逆减法。

语法

op{S}{cond} {Rd,} Rn, Operand2 op{cond} {Rd,} Rn, #imm12; ADD和SUB仅支持

其中:

- ‘*op*’ 属于以下之一：ADD: 加法 ADC: 带进位加法 SUB: 减法 SBC: 带进位减法 RSB: 反向减法
- ‘*S*’ 是一个可选的后缀。如果指定了 S，操作结果的条件码标志会被更新（参见 *Conditional execution on page 65*）
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。如果省略 Rd，目标寄存器是 Rn
- ‘*Rn*’ 是保存第一个操作数的寄存器
- ‘*Operand2*’ 是一个灵活的第二个操作数（参见 *Flexible second operand on page 60* 以获取选项的详细信息）
- ‘*imm12*’ 可以是任意值，在0到4095的范围内

运营

ADD指令将*operand2*或*imm12*的值添加到*Rn*中的值。

ADC指令将Rn和*operand2*中的值相加，以及进位标志。

SUB指令从*Rn*中的值减去*operand2*或imm12的值。

SBC指令从*Rn*中的值减去*operand2*的值。如果进位标志为清零状态，结果会减一。

RSB指令从*operand2*的值中减去*Rn*的值。这很有用，因为*operand2*有广泛的选项范围。

使用ADC和SBC来合成多字节算术运算（参见 *Multiword arithmetic examples on page 84* 和 *ADR on page 70*）

ADDW等同于使用*imm12*操作数的ADD语法。SUBW等同于使用*imm12*操作数的SUB语法。

限制

在这些说明中:

- *Operand2*必须既不是SP也不是PC
- *Rd*只能在ADD和SUB指令中作为SP使用，并且必须满足以下附加限制条件：– *Rn* 也必须是 SP。– 操作数2 的移位操作必须使用LSL限制为最多三位。

- *Rn*只能在ADD和SUB中作为SP。
- *Rd*只能在ADD{cond} PC, PC, *Rm*指令中使用，其中：– 不得指定S后缀。– *Rm*不能是PC或SP。– 如果是条件指令，必须是IT块中的最后一条指令。

- 除ADD{cond} PC, PC, *Rm*指令外， PC, *Rn*只能在ADD和SUB中作为PC使用，并且必须满足以下附加限制：– 不得指定S后缀。– 第二个操作数必须是0到4095范围内的常数。

Note: 1 When using the PC for an addition or a subtraction, bits[1:0] of the PC are rounded to b00 before performing the calculation, making the base address for the calculation word-aligned.

2 If you want to generate the address of an instruction, you have to adjust the constant based on the value of the PC. Arm recommends that you use the ADR instruction instead of ADD or SUB with Rn equal to the PC, because your assembler automatically calculates the correct constant for the ADR instruction.

当*Rd*是PC时，在ADD{cond} PC, PC, *Rm*指令中:

- 写入PC的值的第0位被忽略。
- 当该值的bit[0]被强制置0时，分支跳转到生成的地址。

条件标志

如果指定了S，这些指令会根据结果更新N、Z、C和{v*}标志。

示例

```
ADD R2, R1, R3 SUBS R8, R6, #240 ; 对结果设置标志 RSB R4, R4, #1280 ; 从1280中减去R4的内容 ADCHI R11, R0, R3 ; 仅在C标志置位且Z标志清零时执行
```

多词算术示例

*Specific example 4: 64-bit addition*显示了两条指令，将存储在R2和R3中的64位整数与存储在R0和R1中的另一个64位整数相加，并将结果存储在R4和R5中。

具体示例4：64位加法

```
ADDS R4, R0, R2 ; 将最低有效字相加 ADC R5, R1, R3 ; 带进位的最高有效字相加
```

多字值不需要使用连续的寄存器。*Specific example 5: 96-bit subtraction* 展示了从另一个存储在 R6、R2 和 R8 中的 96 位整数中减去存储在 R9、R1 和 R11 中的 96 位整数的指令。该示例将结果存储在 R6、R9 和 R2 中。

具体示例5：96位减法

```
SUBS R6, R6, R9 ; 减去最低有效字 SBCS R9, R2, R1 ; 减去中间字并带进位 SBC R2, R8, R11 ; 减去最高有效字并带进位
```

3.5.2 AND, ORR, EOR, BIC, and ORN

逻辑与、或、异或、位清除、或非

语法

操作{S}{cond} {Rd,} Rn, 操作数2

其中：

- ‘*op*’ 是以下之一：与：逻辑与. 或：逻辑或或位设置. 异或：逻辑异或. 与非：逻辑与非或位清除. 或非：逻辑或非。
- ‘*S*’ 是一个可选的后缀。如果指定了 S，操作结果上的条件码标志将被更新，参见 *Conditional execution on page 65*。
- ‘*cond*’ 是一个可选的条件码，参见 *Conditional execution on page 65*。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是保存第一个操作数的寄存器。
- ‘*Operand2*’ 是一个灵活的第二个操作数，参见 *Flexible second operand on page 60* 以获取选项的详细信息。

运营

AND、EOR 和 ORR 指令对 *Rn* 和 *operand2* 中的值执行位运算的与、异或和或操作。

BIC指令对*Rn*中的位与*operand2*值中对应位的补码执行按位与操作。

ORN指令对*Rn*中的位与*operand2*值中对应位的取反结果执行OR操作。

限制

不要使用SP或PC。

条件标志

如果S被指定，这些指令：{v*}

- 根据结果更新N和Z标志位。
- 可以在计算*operand2*期间更新C标志，参考*Flexible second operand on page 60*。
- 不要影响V标志。

示例

```
与 R9, R2, #0xFF00     或（仅当相等
） R2, R0, R5     与（带状态标志）R9
, R8, #0x19     异或（带状态标志）R7
, R11, #0x18181818     位清除（R0, R1,
#0xab     或非（R7, R11, R14, 右移 #4
或非（带状态标志） R7, R11, R14,
算术右移 #32
```

3.5.3 自动语音识别，长时记忆，自动语音识别，右移，和 RRX

算术右移、逻辑左移、逻辑右移、右移、以及带扩展的右移。

语法

```
操作{S}{cond} Rd, Rm, Rs 操
作{S}{cond} Rd, Rm, #n RRX{
S}{cond} Rd, Rm
```

其中：

- ‘*op*’ 是以下之一： ASR: 算术右移 LSL: 逻辑左移 LSR: 逻辑右移 ROR: 右移位
- '*S*' 是一个可选的后缀。如果指定了 S，操作结果上的条件码标志会被更新，请参见 *Conditional execution on page 65*。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rm*’ 是保存待移位值的寄存器。
- “*Rs*”是用于保存对值*Rm*进行移位的长度的寄存器。仅使用最低有效字节，其范围为0到255。
- ‘*n*’ 是移位长度。移位长度的范围取决于指令，如下所示： ASR：移位长度从 1 到 32 LSL：移位长度从 0 到 31 LSR：移位长度从 1 到 32 ROR：移位长度从 1 到 31

Note: MOVS Rd, Rm is the preferred syntax for LSLS Rd, Rm, #0.

运营

ASR、LSL、LSR和ROR指令将*Rm*寄存器中的位向左或向右移动，移动的位数由常量*n*或寄存器*Rs*指定。

RRX 将 *Rm* 寄存器中的位向右移动1位。

在所有这些指令中，结果写入到 *Rd*，但 *Rm*寄存器中的值保持不变。不同指令生成的结果详细信息参见 *Shift operations on page 62*。

限制

不要使用SP或PC。

条件标志

如果S被指定:

- 这些指令根据结果更新N和Z标志
- C标志被更新为移出的最后一位，除非移位长度为0（参见 *Shift operations on page 62*）。

示例

```
ASR R7, R8, #9 ; 算术右移9位 LSLS R1, R2, #3 ; 逻辑左移3位并更新标志位 LSR R4, R5, #6 ; 逻辑右移6位 ROR R4, R5, R6 ; 以R6中最低字节的值进行右移 RRX R4, R5 ; 右移并扩展
```

3.5.4 CLZ

统计前导零。

语法

CLZ{cond} Rd, Rm

其中:

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rm*’ 是操作数寄存器。

运营

CLZ指令计算寄存器*Rm*中的值的前导零数量，并将结果返回到*Rd*。如果源寄存器中没有位被设置，结果值为32；如果bit[31]被设置，则为零。

限制

不要使用SP或PC。

条件标志

该指令不会改变标志。

示例

```
CLZ R4,R9 CL
ZNE R2,R3
```

3.5.5 CMP 和 CMN

比较和比较负数。

语法

比较{cond} Rn, 操作数Operand
2 比较不{cond} Rn, 操作数Ope
rand2
其中:

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。”
- ‘*Rn*’ 是存储第一个操作数的寄存器。
- ‘*Operand2*’ 是一个灵活的第二个操作数（参见 *Flexible second operand on page 60* 以获取选项的详细信息）。

运营

这些指令将寄存器中的值与 *operand2* 进行比较。它们根据结果更新条件标志，但不会将结果写入寄存器。

CMP指令将*operand2*的值从*Rn*的值中减去。这与SUBS指令相同，只是结果被丢弃。

CMN指令将*operand2*的值加到*Rn*的值上。这与ADDS指令相同，只是结果被丢弃。

限制

在这些说明中:

- 不要使用PC。
- *Operand2*不得为SP。

条件标志

这些指令根据结果更新 N、Z、C 和 {v*} 标志。

示例

```
比较 R2 和 R9 比较 R0 和 #
6400 比较 SP 与 R7 左移 #2
```

3.5.6 移动和非移动

移动和不移动。

语法

MOV{S}{cond} Rd, 操作数2 MOV
{cond} Rd, #立即数16 MVN{S}{co
nd} Rd, 操作数2

其中：

- ‘*S*’ 是一个可选的后缀。如果指定了 S，操作结果的条件码标志将被更新（参见 *Conditional execution on page 65*）。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Operand2*’ 是一个灵活的第二个操作数（参见 *Flexible second operand on page 60* 以获取选项的详细信息）。
- ‘*imm16*’ 是 0到65535 范围内的任意值。

运营

MOV指令将*operand2*的值复制到*Rd*。

当 MOV 指令中的 *operand2* 是带有除 LSL #0 以外的移位操作的寄存器时，首选语法是相应的移位指令：

- ASR{S}{cond} Rd, Rm, #n 是 MOV{S}{cond} Rd, Rm, ASR #n 的首选语法格式
- LSL{S}{cond} Rd, Rm, #n 是 MOV{S}{cond} Rd, Rm, LSL #n 的推荐语法，如果 n != 0
- LSR{S}{cond} Rd, Rm, #n 是 MOV{S}{cond} Rd, Rm, LSR #n 的首选语法
- ROR{S}{cond} Rd, Rm, #n 是 MOV{S}{cond} Rd, Rm, ROR #n 的首选语法格式
- RRX{S}{cond} Rd, Rm 是 MOV{S}{cond} Rd, Rm, RRX 的首选语法格式

此外，MOV指令允许额外的形式 *operand2* 作为移位指令的同义词：

- MOV{S}{cond} Rd, Rm, ASR Rs 等同于 ASR{S}{cond} Rd, Rm, Rs
- MOV{S}{cond} Rd, Rm, LSL Rs 是 LSL{S}{cond} Rd, Rm, Rs 的同义词
- MOV{S}{cond} Rd, Rm, LSR Rs 是 LSR{S}{cond} Rd, Rm, Rs 的同义指令
- MOV{S}{cond} Rd, Rm, ROR Rs 是 ROR{S}{cond} Rd, Rm, Rs 的同义词

参见 *ASR, LSL, LSR, ROR, and RRX on page 86*。

MVN指令将*operand2*的值取出，对值执行按位逻辑非操作，并将结果存入*Rd*。

Note: The MOVW instruction provides the same function as MOV, but is restricted to use of the imm16 operand.

限制

您只能在MOV指令中使用SP和PC，以下限制条件：

- 第二个操作数必须是寄存器{v*}，不带移位
- 您不得指定 S 后缀

当 *Rd* 是 MOV 指令中的 PC 时：

- 写入程序计数器的值的位[0]被忽略
- 当该值的bit[0]被强制置0时，分支跳转到生成的地址。

Note: Though it is possible to use MOV as a branch instruction, Arm strongly recommends the use of a BX or BLX instruction to branch for software portability to the Arm instruction set.

条件标志

如果S被指定，这些指令：{v*}

- 根据结果更新N和Z标志
- 可以在计算*operand2*时更新C标志（参见*Flexible second operand on page 60*）。
- 不要影响V标志

示例

MOVS R11, #0x000B ; 将0x000B的值写入R11，标志位会被更新 MOV R1, #0xFA05 ; 将0xFA05的值写入R1，标志位不会被更新 MOVS R10, R12 ; 将R12中的值写入R10，标志位会被更新 MOV R3, #23 ; 将23的值写入R3 MOV R8, SP ; 将堆栈指针的值写入R8 MVNS R2, #0xF ; 将0xFFFFFFF0（0xF的按位取反）的值写入R2并更新标志位

3.5.7 移动

移到顶部

语法

MOVT{cond} Rd, #imm16

其中:

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。”
- ‘*Rd*’ 是目标寄存器。
- ‘*imm16*’ 是一个 16 位立即数常量。

运营

MOVT将一个16位的立即数*imm16*写入其目标寄存器的高半字*Rd*[31:16]。该写入操作不会影响*Rd*[15:0]。

MOV，M OVT指令对使您能够生成任意32位 {v*} 常数

限制

*Rd*必须既不是SP也不是PC。

条件标志

该指令不会改变标志。

示例

```
MOVT R3, #0xF123 ; 将 0xF123 写入 R3 的上半字，下半字和程序状态寄存器保持
不变
```

3.5.8 REV, REV16, REVSH 和 RBIT

反转字节和反转位

语法

操作{cond} Rd, Rn

其中:

- ‘*op*’ 是以下之一：REV：在字中反转字节顺序。REV16：每个半字独立地反转字节顺序。REVSH：反转底部半字的字节顺序，并符号扩展到32位。RBIT：反转32位字中的位顺序。
- ‘*cond*’ 是一个可选的状态码，参见 *Conditional execution on page 65*。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是保存操作数的寄存器。

运营

使用这些说明来更改数据的字节序:

- REV: 转换以下任一类型：– 32位大端格式数据为小端格式数据 – 或 32位小端格式数据为大端格式数据。
- REV16：将以下内容转换：– 将16位大端数据转换为小端数据 – 或将16位小端数据转换为大端数据。
- REVSH：将以下内容转换：– 16位有符号大端序数据转换为32位有符号小端序数据 – 或 16位有符号小端序数据转换为32位有符号大端序数据。

限制

不要使用SP或PC。

条件标志

这些指令不会更改标志。

示例

```
REV R3, R7 ; 反转 R7 中值的字节顺序并写入 R3 REV16 R0, R0 ; 反转 R0 中每个16位半字的字节顺序 REVSH R0, R5 ; 反转带符号的半字 REVHS R3, R7 ; 反转（高位或相同条件） RBIT R7, R8 ; 反转 R8 中值的位顺序并写入 R7
```

3.5.9 SADD16 和 SADD8

带符号加16和带符号加8

语法

操作{cond}{Rd,} Rn, Rm

其中:

- op 可以是以下任意一种：SADD16: 执行两个 16 位带符号整数加法运算。SADD8: 执行四个 8 位带符号整数加法运算。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是保存操作数的寄存器。
- ‘*Rm*’ 是保存操作数的第二个寄存器。

运营

使用 i这些指令用于在 p 中执行半字或字节的加法 并行:

SADD16指令:
1. 将第一个操作数的每个半字加到第二个操作数的相应半字上。2. 将结果写入目标寄存器的相应半字中。

SADD8指令:
1. 将第一个操作数的每个字节与第二个操作数的对应字节相加。 2. 将结果写入目标寄存器的对应字节中。

限制

请勿使用SP和PC。

条件标志

这些指令不会更改标志。

示例

SADD16 R1, R0 ; 将 R0 中的半字与 R1 对应的半字相加，并写入 R1 的对应半字。SADD8 R4, R0, R5 ; 将 R0 中的字节与 R5 对应的字节相加，并写入 R4 的对应字节。

3.5.10 SHADD16 和 SHADD8

带符号半加16和带符号半加8

语法

操作{cond}{Rd,} Rn, Rm

其中:

- op 可以是以下任意一种：SHADD1
  6：有符号整数半加 16。SHADD8
  ：有符号整数半加 8。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是保存操作数的寄存器。
- ‘*Rm*’ 是第二个操作数寄存器。

运营

使用这些指令来添加16位和8位数据，然后将结果除以二，并在写入目标寄存器之前完成:

SHADD16指令:
1. 将第一个操作数的每个半字与其对应的第二个操作数的半字相加。 2. 将结果向右移动一位，使数据减半。 3. 将半字结果写入目标寄存器。

SHADDB8指令:
1. 将第一个操作数的每个字节与第二个操作数的对应字节相加。 2. 将结果向右移动一位，将数据减半。 3. 将字节结果写入目标寄存器。

限制

请勿使用SP和PC。

条件标志

这些指令不会更改标志。

示例

SHADD16 R1, R0 ; 将R0中的半字加到R1对应的半字上 & ; 将结果的一半写入R1对应的半字中SHADD8 R4, R0, R5 ; 将R0中的字节加到R5对应的字节上和 ; 将结果的一半写入R4对应的字节中

3.5.11 SHASX 和 SHSAX

带符号的加减运算（二分之一）并交换 / 带符号的减加运算（二分之一）并交换

语法

操作{cond} {Rd}, Rn, Rm

其中:

- op 是以下任意一种：SHASX：加减操作，带交换和减半。 SHSAX：减加操作，带交换和减半。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）：
- ‘*Rd*’ 是目标寄存器:
- ‘*Rn*’ 是保存操作数的寄存器:
- “*Rn*”、”*Rm*“ 是存储第一个和第二个操作数的寄存器。

运营

SHASX指令:

1. 将第一个操作数的高字与第二个操作数的低字相加。2. 将加法的半字结果写入目标寄存器的高字，向右移位一位，导致除以二或减半。3. 从第一个操作数的低字中减去第二个操作数的高字。4. 将除法的半字结果写入目标寄存器的低字，向右移位一位，导致除以二或减半。

SHSAX指令:

1. 从第一个操作数的高字上半部分减去第二个操作数的低字下半部分。2. 将加法的半字结果写入目标寄存器的低字下半部分，右移一位，导致除以二或减半。3. 将第一个操作数的低字下半部分与第二个操作数的高字上半部分相加。4. 将除法的半字结果写入目标寄存器的高字上半部分，右移一位，导致除以二或减半。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

示例

SHASX R7, R4, R2 ; 将 R4 的高半字加到 R2 的低半字上 ; 并将结果的高半字写入 R7 的高半字 ; 从 R2 的低半字中减去高半字

; R4 和将半结果写入 R7 的低半字
SHSAX R0, R3, R5 ; 将 R5 的低半字从 R3 的高半字中减去，并将结果的一半写入 R0 的高半字；将 R5 的高半字加到 R3 的低半字上，并将结果的一半写入 R0 的低半字。

3.5.12 SHSUB16 和 SHSUB8

带符号的减半减16和带符号的减半减 t 八

语法

操作{cond}{Rd,} Rn, Rm

其中:

- op 可以是以下任意一种：SHSUB16: 带符号的减半减法 16 SHSUB8: 带符号的减半减法 8
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器
- ‘*Rn*’ 是存储操作数的寄存器
- ‘*Rm*’ 是第二个操作数寄存器

运营

使用这些指令来添加16位和8位数据，然后将结果除以二，并在写入目标寄存器之前完成:

SHSUB16指令:
1. 将第二个操作数的每个半字从第一个操作数的相应半字中减去。 2. 将结果右移一位，将数据减半。 3. 将减半后的半字结果写入目标寄存器。

SHSUBB8指令:
1. 将第二个操作数的每个字节从第一个操作数的对应字节中减去， 2. 将结果向右移一位位（bit），使数据减半， 3. 将对应的有符号字节结果写入目标寄存器。

限制

请勿使用SP和PC。

条件标志

这些指令不会更改标志。

示例

SHSUB16 R1, R0 ; 从R1对应的半字中减去R0中的半字，并写入R1对应的半字 SHSUB8 R4, R0, R5 ; 从R5对应的字节中减去R0中的字节

; 并写入R4中的对应字节。

3.5.13 SSUB16和SSUB8

带符号减法16和带符号减法8

语法

操作{cond}{Rd,} Rn, Rm

其中:

- op 是以下之一：SSUB16：执行两个16位带符号整数减法运算。SSUB8：执行四个8位带符号整数减法运算。

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是保存操作数的寄存器。
- ‘*Rm*’ 是第二个操作数寄存器。

运营

使用这些说明来更改数据的字节序:

SSUB16指令:
1. 将第二个操作数的每个半字从第一个操作数的相应半字中减去。 2. 将两个带符号半字的差值写入目标寄存器的相应半字中。

SSUB8指令:
1. 将第二个操作数的每个字节从第一个操作数的对应字节中减去。2. 将四个带符号字节的差值结果写入目标寄存器的对应字节中。

限制

请勿使用SP和PC。

条件标志

这些指令不会更改标志。

示例

```
SSUB16 R1,  R0     ; 从对应的半字中减去 R0 中的半字
                   ; R1 的并写入对应的半字 R1
SSUB8 R4,   R0,  R5 ; 从对应字节中减去 R5 的字节
                   ; R0，并将对应的字节写入R4。
```

3.5.14 SASX 和 SSAX

带符号加减带交换和带符号减加带交换。

语法

op{cond} {Rd}, Rm, Rn

其中:

- op 是以下之一：SASX：带交换的有符号加减。
  SSAX：带交换的有符号减加。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 和 ‘*Rm*’ 是保存第一个和第二个操作数的寄存器。

运营

SASX指令:
1. 将第一个操作数的有符号的高半字与第二个操作数的有符号的低半字相加。2. 将加法的有符号结果写入目标寄存器的高半字。3. 从第一个操作数的有符号的高半字中减去第二个操作数的有符号的低半字。4. 将减法的有符号结果写入目标寄存器的低半字。

SSAX指令:
1. 将第一个操作数的有符号高字减去第二个操作数的有符号半字。 2. 将加法的有符号结果写入目标寄存器的底部半字。 3. 将第一个操作数的有符号高字与第二个操作数的有符号半字相加。 4. 将减法的有符号结果写入目标寄存器的顶部半字。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

示例

SASX R0 , R4, R5 ; 将 R4 的高半字加到 R5 的低半字，并写入 R0 的高半字； 将 R5 的低半字从 R4 的高半字中减去，并写入 R0 的低半字

SSAX R7 , R3, R2 ; 从 R3 的低半字中减去 R2 的高半字；并将结果写入 R7 的低半字

; 将 R3 的高字半与 R2 的低字半相加，并 ; 写入 R7 的高字半。

3.5.15 测试和等价

测试位和测试等效性。

语法

TST{cond} Rn, 操作数2 TEQ{c
ond} Rn, 操作数2

其中:

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。”
- ‘*Rn*’ 是存储第一个操作数的寄存器。
- ‘*Operand2*’ 是一个灵活的第二个操作数（参见 *Flexible second operand on page 60* 以获取选项的详细信息）。

运营

这些指令将寄存器中的值与 *operand2* 进行比较。它们根据比较结果更新条件标志，但不将结果写入寄存器。

TST指令对*Rn*中的值和*operand2*的值执行按位与操作。这与ANDS指令相同，但会丢弃结果。

要测试 *Rn* 中的某一位是 0 还是 1，可以使用将该位设置为 1、其余所有位清零的 *operand2* 常量与 TST 指令结合。

TEQ指令对*Rn*中的值和*operand2*的值执行按位异或操作。这与EORS指令相同，只是它会丢弃结果。

使用TEQ指令来测试两个值是否相等，而不影响V或C标志。

TEQ 也可以用于测试值的符号。比较之后，N 标志位是两个操作数符号位的逻辑异或结果。

限制

不要使用SP或PC。

条件标志

这些说明:

- 根据结果更新N和Z标志
- 可以在计算 *operand2* 期间更新 C 标志（参见 *Flexible second operand on page 60*）。
- 不要影响V标志

示例

```
测试 R0, #0x3F8 ; 对R0的值与0x3F8执行按位与操作， ; APSR会被更新，但结果被
                  丢弃
TEQEQ R10, R9 ; 条件判断 R10 中的值是否等于 ; R9 中的值，APSR 被更新但结果被丢弃
```

3.5.16 UADD16和UADD8

无符号加法16 和 无符号加法8

语法

操作{cond}{Rd,} Rn, Rm

其中:

- op 是以下之一：UADD16：执行两个16位无符号整数加法
  。UADD8：执行四个8位无符号整数加法。

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是第一个保存操作数的寄存器。
- ‘*Rm*’ 是保存操作数的第二个寄存器。

运营

使用这些指令来添加16位和8位无符号数据。 这是一个测试 {v*}。

UADD16指令:

1. 将第一个操作数的每个半字加到第二个操作数的对应半字上
   操作数。2。将无符号结果写入目标寄存器的相应半字。

UADD16指令:

1. 将第一个操作数的每个字节与第二个操作数的对应字节相加。 2. 将无符号结果写入目标寄存器的对应字节中。

限制

请勿使用SP和PC。

条件标志

这些指令不会更改标志。

示例

UADD16 R1, R0 ; 将R0中的半字加到R1对应的半字上, ; 写入R1对应的半字 UADD8 R4, R0, R5 ; 将R0中的字节加到R5对应的字节上, ; 写入R4对应的字节

3.5.17 UASX 和 USAX

加减法带借位和减加法带进位

语法

操作{cond} {Rd}, Rn, Rm

其中:

- op 是其中之一： UASX：加减带交换。
  USAX：减加带交换。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 和 ‘*Rm*’ 是保存第一个和第二个操作数的寄存器。

运营

UASX指令:
1. 将第二个操作数的高半字从第一个操作数的低半字中减去。2. 将减法得到的无符号结果写入目标寄存器的低半字。3. 将第一个操作数的高半字与第二个操作数的低半字相加。4. 将加法得到的无符号结果写入目标寄存器的高半字。

USAX指令:
1. 将第一个操作数的低半字加到第二个操作数的高半字上。2. 将加法的无符号结果写入目标寄存器的低半字。3. 将第二个操作数的低半字从第一个操作数的高半字中减去。4. 将减法的无符号结果写入目标寄存器的高半字。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

示例

UASX R0, R4, R5 ; 将 R4 的高半字与 R5 的低半字相加，并写入 R0 的高半字；将 R5 的低半字从 R0 的高半字中减去，并写入 R0 的低半字 USAX R7, R3, R2 ; 将 R2 的高半字从 R3 的低半字中减去，并写入 R7 的低半字；将 R3 的高半字与 R2 的低半字相加，并

; 写入 R7 的高半字

3.5.18 UHADD16 和 UHADD8

无符号半加16和无符号半加8

语法

操作{cond}{Rd,} Rn, Rm

其中:

- op 是以下之一：UHADD16: 无符号舍入加法 16。UHADD8: 无符号舍入加法 8。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是保存第一个操作数的寄存器。
- ‘*Rm*’ 是存储第二个操作数的寄存器。

运营

使用这些指令将16位和8位数据相加，然后将结果除以二，再将结果写入目标寄存器:

UHADD16指令:

1. 将第一个操作数的每个半字加到第二个操作数的对应半字上
操作数。将半字结果向右移动一位，将数据减半。将无符号结果写入目标寄存器中的对应半字位置。{v*}

UHADD8指令:

1. 将第一个操作数的每个字节与其对应字节的第二个操作数相加。 2. 将字节结果向右移动一位，将数据减半。 3. 在目标寄存器的对应字节中写入无符号结果。

限制

请勿使用SP和PC。

条件标志

这些指令不会更改标志。

示例

UHADD16 R7, R3 ; 将R7中的半字与R3中的对应半字相加，并将结果的一半写入R7中的对应半字 UHADD8 R4, R0, R5 ; 将R0中的字节与R5中的对应字节相加，并将结果的一半写入R4中的对应字节

3.5.19 UHASX 和 UHSAX

无符号半加减法带进位和无符号半减加法带进位

语法

操作{cond} {Rd}, Rn, Rm

其中:

- op 是以下之一：UHASX: 加减带交换和减半。UHSAX: 减加带交换和减半。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ ‘*Rm*’ 是存储第一个和第二个操作数的寄存器。

运营

UHASX指令:

1. 将第一个操作数的高半字与第二个操作数的低半字相加。2. 将结果向右移一位，导致除以二，或减半。3. 将加法的半字结果写入目标寄存器的高半字。4. 从第一个操作数的低半字减去第二个操作数的高半字。5. 将结果向右移一位，导致除以二，或减半。6. 将除法的半字结果写入目标寄存器的低半字。

UHSAX指令:

1. 从第一个操作数的高字中减去第二个操作数的低字。2. 将结果向右移位一位，导致除以二，或减半。3. 将减法的半字结果写入目标寄存器的高字部分。4. 将第一个操作数的低字部分与第二个操作数的高字部分相加。5. 将结果向右移位一位，导致除以二，或减半。6. 将加法的半字结果写入目标寄存器的低字部分。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

示例

```
UHASX R7, R4, R2        ; 将R4的高半字与R2的低半字相加; 并将结果的一半写入R7的高半字; 从R7
                          的低半字中减去R2的高半字; 并将结果的一半写入R7的低半字
```

UHSAX R0, R3, R5 ; 将 R5 的低半字从 R3 的高半字中减去，并将结果的一半写入 R0 的高半字；将 R5 的高半字加到底部半字的 R3 上，并将结果的一半写入 R0 的低半字。

3.5.20 UHSUB16 和 UHSUB8

无符号减半减法16 和 无符号减半减法8

语法

操作{cond}{Rd,} Rn, Rm

其中：

- op 是以下任意一种：UHSUB16：执行两个无符号16位整数加法，将结果除以二，并将结果写入目标寄存器。UHSUB8：执行四个无符号8位整数加法，将结果除以二，并将结果写入目标寄存器。

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是第一个保存操作数的寄存器。
- ‘*Rm*’ 是保存操作数的第二个寄存器。

运营

使用这些指令来添加16位和8位数据，然后将结果除以二，并在写入目标寄存器之前完成：

UHSUB16 指令：
1. 将第二个操作数的每个半字从第一个操作数的相应半字中减去。2. 将每个半字结果向右移位一位，将数据减半。3. 将每个无符号半字结果写入目标寄存器的相应半字位置。

UHSUB8指令：
1. 将第二个操作数的每个字节从第一个操作数的对应字节中减去。2. 将每个字节结果右移一位，将数据减半。3. 将无符号字节结果写入目标寄存器的对应字节。

限制

请勿使用SP和PC。

条件标志

这些指令不会更改标志。

示例

UHSUB16 R1, R0 ; 从R0中的半字减去R1对应的半字；并将减半的结果写入R1对应的半字中

UHSUB8 R4, R0, R5 ; 从 R0 中对应的字节减去 R5 的字节，并将结果的一半写入 R4 中对应的字节。

3.5.21 选择

选择字节。根据GE标志的值，从其第一个操作数或第二个操作数中选取其结果的每个字节。

语法

选择{<乘>}{<除>} {<读取>,} <寄存器编号>, <寄存器内存>

其中:

- **<c>，<q>** 是标准汇编语法字段。
- **<Rd>** 是目标寄存器。
- **<Rn>** 是第一个操作数寄存器。
- **<Rm>** 是第二个操作数寄存器。

运营

SEL指令:

1. 读取APSR.GE中每一位的值。 2. 根据APSR.GE的值，将目标寄存器赋值为第一个或第二个{v*}的值。

限制

无。

条件标志

这些指令不会更改标志。

示例

SADD16 R0, R1, R2 ; 根据结果设置GE位 SEL R0, R0, R3 ; 根据GE选择R0或R3中的字节

3.5.22 USAD8

绝对差之和

语法

USAD8{条件码}{Rd,} Rn, Rm

其中:

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。”
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是第一个操作数寄存器。
- ‘*Rm*’ 是第二个操作数寄存器。

运营

USAD8指令:
1. 将第二个操作数寄存器的每个字节从第一个操作数寄存器的对应字节中减去。 2. 将差值的绝对值相加。 3. 将结果写入目标寄存器。

限制

不要使用SP，而应使用PC。

条件标志

这些指令不会更改标志。

示例

```
USAD8 R1, R4, R0      ; 从R0中的每个字节减去对应字节的
                      ; R4 将差值相加并写入 R1
USAD8 R0, R5          从 R0 的对应字节中减去 R5 的字节,
                      ; 将差值相加并写入 R0。
```

3.5.23 USADA8

无符号绝对差值的总和与累积

语法

USADA8{cond}{Rd,} Rn, Rm, Ra

其中:

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。”
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是第一个操作数寄存器。
- ‘*Rm*’ 是第二个操作数寄存器。
- ‘*Ra*’ 是包含累加值的寄存器。

运营

USADA8 指令:
1. 将第二个操作数寄存器的每个字节从第一个操作数寄存器的对应字节中减去。2. 将无符号绝对差值相加。3. 将累加值加到绝对差值的总和中。4. 将结果写入目标寄存器。{v*}

限制

请勿使用SP和PC。

条件标志

这些指令不会更改标志。

示例

USADA8 R1, R0, R6 ; 从R0中的字节减去R1对应的半字 ; 将差值相加，加上R6的值，写入R1 USADA8 R 4, R0, R5, R2 ; 从R0对应的字节减去R5中的字节 ; 将差值相加，加上R2的值，写入R4.

3.5.24 USUB16和USUB8

无符号减法16和无符号减法8

语法

操作{cond}{Rd,} Rn, Rm

其中：

- op 是以下任意一种：USUB16：
  Unsigned Subtract 16。USUB8：
  Unsigned Subtract 8。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器
- ‘*Rn*’ 是存储操作数的寄存器
- ‘*Rm*’ 是第二个操作数寄存器

运营

使用这些指令在将结果写入目标寄存器之前进行16位和8位数据的减法运算：

USUB16指令：
1. 将第二个操作数寄存器中的每个半字从第一个操作数寄存器的对应半字中减去。 2. 将无符号结果写入目标寄存器的对应半字中。

USUB8指令：
1. 将第二个操作数寄存器的每个字节从第一个操作数寄存器的对应字节中减去。 2. 将无符号字节结果写入目标寄存器的对应字节中。

限制

不要使用SP或PC。

条件标志

这些指令不会更改标志。

示例

USUB16 R1, R0 ; 从 R0 中的半字减去 对应的半字； R1 并写入 R1 对应的半字。USUB8 R4, R0, R5 ; 从 R0 中对应的字节减去 R5 的字节，并。

; 写入到R4中的对应字节。

3.6 乘法和除法指令

*Table 29*显示乘法和除法指令。

表29. 乘法和除法指令

Mnemonic	Brief description	See
MLA	Multiply with Accumulate, 32-bit result	*MUL, MLA, and MLS on page 110*
MLS	Multiply and Subtract, 32-bit result	*MUL, MLA, and MLS on page 110*
MUL	Multiply, 32-bit result	*MUL, MLA, and MLS on page 110*
SDIV	Signed Divide	*SDIV and UDIV on page 124*
SMLA[B,T]	Signed Multiply Accumulate (halfwords)	*SMLA and SMLAW on page 112*
SMLAD, SMLADX	Signed Multiply Accumulate dual	*SMLAD on page 114*
SMLAL	Signed Multiply with Accumulate (32x32+64), 64-bit result	*SMLAL and SMLALD on page 115*
SMLAL[B,T]	Signed Multiply Accumulate Long (halfwords)	*SMLAL and SMLALD on page 115*
SMLALD, SMLALDX	Signed Multiply Accumulate Long Dual	*SMLAL and SMLALD on page 115*
SMLAW[B|T]	Signed Multiply Accumulate (word by halfword)	*SMLA and SMLAW on page 112*
SMLSD	Signed Multiply Subtract Dual	*SMLSD and SMLSLD on page 117*
SMLSLD	Signed Multiply Subtract Long Dual	*SMLSD and SMLSLD on page 117*
SMMLA	Signed Most Significant Word Multiply Accumulate	*SMMLA and SMMLS on page 119*
SMMLS, SMMLSR	Signed Most Significant Word Multiply Subtract	*SMMLA and SMMLS on page 119*
SMUAD, SMUADX	Signed dual multiply add	*SMUAD and SMUSD on page 121*
SMUL[B,T]	Signed multiply (word by halfword)	*SMUL and SMULW on page 122*
SMMUL, SMMULR	Signed most significant word multiply	*SMMUL on page 120*
SMULL	Signed multiply (32x32), 64-bit result	*SMMUL on page 120*
SMULWB, SMULWT	Signed multiply (word by halfword)	*SMUL and SMULW on page 122*
SMUSD, SMUSDX	Signed dual multiply subtract	*SMUAD and SMUSD on page 121*
UDIV	Unsigned Divide	*SMLA and SMLAW on page 112*
UMAAL	Unsigned Multiply Accumulate Accumulate Long (32x32+32+32), 64-bit result	*UMULL, UMAAL and UMLAL on page 111*
UMLAL	Unsigned Multiply with Accumulate (32x32+64), 64-bit result	*UMULL, UMAAL and UMLAL on page 111*
UMULL	Unsigned Multiply (32x32), 64-bit result	*UMULL, UMAAL and UMLAL on page 111*

3.6.1 乘法、乘加、乘减

乘法、乘法累加、乘法减法，使用32位操作数，产生32位结果。

语法

MUL{S}{cond} {Rd,} Rn, Rm ; 乘法MLA{cond} Rd, Rn, Rm, Ra ; 乘法并累加MLS{cond} Rd, Rn, Rm, Ra ; 乘法并相减

其中:

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。”
- ‘*S*’ 是一个可选的后缀。如果指定了 S，则在操作结果上更新状态码标志（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。如果省略 *Rd*，则目标寄存器为 *Rn*.。
- ‘*Rn*’ 、 ‘*Rm*’是保存待相乘值的寄存器。
- ‘*Ra*’ 是一个保存的值的寄存器，用于加到或减去。

运营

MUL指令将来自*Rn*和*Rm*的值相乘，并将结果的最低有效32位存入*Rd*。

MLA指令将*Rn*和*Rm*的值相乘，加上*Ra*的值，并将结果的最低有效32位存储到*Rd*中。

MLS指令将从*Rn*和*Rm*中获取的值相乘，将乘积从从*Ra*中获取的值中减去，并将结果的最低有效32位存入*Rd*中。

结果不依赖于操作数是有符号的还是无符号的。

限制

在这些指令中，请勿使用SP或PC。

如果您在使用MUL指令时使用S后缀:

- *Rd*， *Rn*， 和 *Rm* 都必须位于范围 R0到R7
- *Rd*必须与 *Rm* 相同
- 您不得使用 *cond* 后缀

条件标志

如果S被指定，MUL指令:

- 根据结果更新N和Z标志。
- 不影响C和V标志。

示例

```
乘法 R10, R2, R5                ; 乘法, R10 = R2 x R5
现代语言协会 R10, R2, R1, R5    ; 乘法累加，R10 = (R2 × R1) + R5
带符号乘法 R0, R2, R2           ; 乘法标志位更新，R0 = R2 x R2
MULLT R2, R3, R2                ; 有条件地乘法，R2 = R3 x R2
多层系统 R4, R5, R6, R7         ; 乘法后减法，R4 = R7 - (R5 x R6)
```

3.6.2 UMULL, UMAAL 和 UMLAL

无符号长乘法，可选累加，32位操作数，产生64位结果。{v*}

语法

操作{cond} RdLo, RdHi, Rn, Rm

其中:

- ‘*op*’ 是以下之一：UMULL：无符号长乘法。 UMAAL：无符号长乘法，带累积累积。 UMLAL：无符号长乘法，带累积

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- “*RdHi, RdLo*”是目标寄存器。它们还存储累加值。
- ‘*Rn, Rm*’ 是保存第一个和第二个操作数的寄存器。

运营

UMULL指令:

1. 将第一个操作数和第二个操作数中的两个无符号整数相乘。 2. 将结果的最低有效32位写入 *RdLo*。 3. 将结果的最高有效32位写入 *RdHi*。

UMAAL指令:

1. 将第一个和第二个操作数中的两个无符号32位整数相乘。 2. 将*RdHi*中的无符号32位整数加到乘法的64位结果中。 3. 将*RdLo*中的无符号32位整数加到加法的64位结果中。 4. 将结果的高32位写入*RdHi*。 5. 将结果的低32位写入*RdLo*。

UMLAL指令:

1. 第一个操作数和第二个操作数中的两个无符号整数相乘。2. 将64位结果与存储在*RdHi*和*RdLo*中的64位无符号整数相加。3. 将结果写回*RdHi*和*RdLo*。

限制

在这些说明中:

- 不要使用SP或PC。
- *RdHi*并且 *RdLo* 必须为不同的寄存器。

条件标志

这些指令不会影响状态标志。

示例

UMULL R0, R4, R5, R6 ; 将R5和R6相乘，将高32位写入R4；将低32位写入R0 UMAAL R3, R6, R2, R7 ; 将R2和R7相乘，将R6相加，将R3相加，写入

; 将高32位传送到R6，将低32位传送到R3 UMLAL R2, R1, R3, R5 ; 将R5和R3相乘，将R1:R2相加，结果写入R1:R2.

3.6.3 带符号乘加和带符号乘加带进位

带符号乘积累加（半字）

语法

操作{XY}{cond} Rd, Rn, Rm 操作{Y}{cond} Rd, Rn, Rm, Ra

哪里

- op 是以下之一：SMLA：带符号的乘积累加长（半字）。X 和 Y 指定源寄存器 *Rn* 和 *Rm* 的哪一半作为第一个和第二个乘法操作数。– 如果 X 是 B，则使用 *Rn* 的低半字，位 [15:0]。– 如果 X 是 T，则使用 *Rn* 的高半字，位 [31:16]。– 如果 Y 是 B，则使用 *Rm* 的低半字，位 [15:0]。– 如果 Y 是 T，则使用 *Rm* 的高半字，位 [31:16]。SMLAW：带符号的乘积累加（字与半字）。Y 指定源寄存器 *Rm* 的哪一半作为第二个乘法操作数。– 如果 Y 是 T，则使用 *Rm* 的高半字，位 [31:16]。– 如果 Y 是 B，则使用 *Rm* 的低半字，位 [15:0]。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- "*Rd*"是目标寄存器。如果省略*Rd*，则目标寄存器为*Rn.*
- “*Rn*”和“*Rm*”是保存待相乘值的寄存器。
- ‘*Ra*’ 是一个寄存器，保存将要加到或减去的值。

运营

SMALBB、SMLABT、SMLATB、SMLATT指令：
1. 从 *Rn* 和 *Rm* 中乘以指定的有符号半字，上半部分或下半部分的值。2. 将 *Ra* 中的值加到得到的32位乘积中。3. 将乘法和加法的结果写入 Rd。4. 源寄存器中未指定的半字将被忽略。

SMLAWB 和 SMLAWT 指令：
1. 将 *Rn* 中的 32 位带符号值乘以：a) *Rm* 的高带符号半字，T 指令后缀。b) *Rm* 的低带符号半字，B 指令后缀。2. 将 *Ra* 中的 32 位带符号值加到 48 位乘积的高 32 位上。3. 将乘法和加法的结果写入 Rd。4. 48 位乘积的低 16 位被忽略。5. 如果在累加值相加过程中发生溢出，指令会将 Q 标志设置在 APSR 中。乘法过程中不会发生溢出。

限制

在这些说明中，不要使用 SP 或 PC。

条件标志

如果发生溢出，Q标志会被设置。

示例

SMLABB R5, R6, R4, R1 ；将R6和R4的低半字相乘，加上 ; R1，并将结果写入R5 SMLATB R5, R6, R4, R1 ；将R6的高半字与R4的低半字相乘，加上R1，并将结果写入R5 SMLATT R5, R6, R4, R1 ；将R6和R4的高半字相乘，加上 ; R1，并将总和写入R5 SMLABT R5, R6, R4, R1 ；将R6的低半字与R4的高半字相乘，加上R1，并将结果写入R5 SMLABT R4, R3, R2 ；将R4的低半字与R3的高半字相乘，加上R2，并将结果写入R4 SMLAWB R10, R2, R5, R3 ; 将R2与R5的低半字相乘，加上 ; R3的结果，并将高32位写入R10 SMLAWT R10, R2, R1, R5 ; 将R2与R1的高半字相乘，加上R5 ; 并将高32位写入R10.

3.6.4 SMLAD

带符号乘积累加长双精度

语法

操作{X}{cond} Rd, Rn, Rm, Ra ；

其中:

- op 是以下之一：SMLAD: 带符号乘法累加双操作数。SMLADX: 带符号乘法累加双操作数反向。X 指定源寄存器 *Rn* 中哪一个半字作为乘法操作数。若省略 X，则乘法使用底 × 底和顶 × 顶。若存在 X，则乘法使用底 × 顶和顶 × 底。

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是第一个操作数寄存器，存储待相乘的值。
- “‘*Rm*’ 是第二个操作数寄存器。”
- ‘*Ra*’ 是累计值。

运营

SMLAD 和 SMLADX 指令将两个操作数视为四个半字 16 位值。 SMLAD 和 SMLADX 指令:

1. 或者:

   a) 如果X不存在，则将*Rn*中的高有符号半字值与*Rm*中的高有符号半字相乘，以及将*Rn*中的低有符号半字值与*Rm*中的低有符号半字相乘。b) 如果X存在，则将*Rn*中的高有符号半字值与*Rm*中的低有符号半字相乘，以及将*Rn*中的低有符号半字值与*Rm*中的高有符号半字相乘。

2. 将两个乘法结果相加，并将结果存入 *Ra* 中的有符号的32位值。
3. 将乘法和加法的32位有符号结果写入 Rd。

限制

不要使用SP或PC。

条件标志

这些指令不会更改标志。

示例

SMLAD R10, R2, R1, R5 ; 将R2中的两个半字值与R1中的对应半字相乘，加上R5并将结果写入R10 {v*}

SMLALDX R0, R2, R4, R6 ；将R2的高半字与R4的低半字相乘，将R2的低半字与R4的高半字相乘，将结果与R6相加并写入R0。

3.6.5 缩放累加和缩放累加双精度

带符号乘积累加长整数，带符号乘积累加长整数（半字）和带符号乘积累加长整数双。

语法

操作{cond} RdLo, RdHi, Rn, Rm 操作{XY}
{cond} RdLo, RdHi, Rn, Rm 操作{X}{cond}
RdLo, RdHi, Rn, Rm

其中:

- op 是以下之一：– SMLAL: 带符号的乘法累加长。– SMLAL: 带符号的乘法累加长（半字，X 和 Y）。X 和 Y 指定源寄存器 *Rn* 和 *Rm* 中的哪个半字用作第一个和第二个乘法操作数：如果 X 是 B，则使用 *Rn* 的底半字，即位 [15:0]。如果 X 是 T，则使用 *Rn* 的顶半字，即位 [31:16]。如果 Y 是 B，则使用 *Rm* 的底半字，即位 [15:0]。如果 Y 是 T，则使用 *Rm* 的顶半字，即位 [31:16]。– SMLALD: 带符号的乘法累加长双。– SMLALDX: 带符号的乘法累加长双反向：如果省略 X，则乘法运算使用底 × 底和顶 × 顶。如果存在 X，则乘法运算使用底 × 顶和顶 × 底。

- “‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）”
- ‘*RdHi, RdLo*’是目标寄存器。*RdLo*是64位整数的低32位，*RdHi*是高32位。对于SMLAL、SMLALBB、SMLALBT、SMLALTB、SMLALTT、SMLALD和SMLALDX，它们还保存累加值。
- ‘*Rn*’、‘*Rm*’是存储第一个和第二个操作数的寄存器

运营

SMLAL指令:
1. 将*Rn*和*Rm*中的二进制补码有符号字值相乘。2. 将*RdLo*和*RdHi*中的64位值相加到结果64位乘积中。3. 将乘法和加法得到的64位结果写入*RdLo*和*RdHi*。

SMLALBB、SMLALBT、SMLALTB 和 SMLALTT 指令:
1. 乘以指定的有符号半字上半部分或下半部分的值，来自 *Rn* 和 *Rm*。 2. 将符号扩展的32位乘积相加到 *RdLo* 和 *RdHi* 中的64位值。 3. 将乘法和加法的64位结果写入 *RdLo* 和 *RdHi*。

源寄存器中未指定的半字被忽略。

SMLALD 和 SMLALDX 指令将 *Rn* 和 *Rm* 中的值解释为四个半字的二进制补码有符号的16位整数。这些指令:

- 如果X不存在，将*Rn*的高有符号半字值与*Rm*的高有符号半字相乘，并将*Rn*的低有符号半字值与*Rm*的低有符号半字相乘。
- 或者如果X存在，将*Rn*的高有符号半字值与*Rm*的低有符号半字相乘，并将*Rn*的低有符号半字值与*Rm*的高有符号半字相乘。
- 将两个乘法结果相加，得到带符号的64位值在*RdLo*和*RdHi*中的结果，从而形成最终的64位乘积。
- 将64位产品写入 *RdLo* 和 *RdHi* 中。

限制

在这些说明中:

不要使用SP或PC。

*RdHi*和 *RdLo* 必须是不同的寄存器。

条件标志

这些指令不会影响状态标志。

示例

SMLAL R4, R5, R3, R8 ; 将 R3 和 R8 相乘，将 R5:R4 相加并写入 ; R5:R4 SMLALBT R2, R1, R6, R7 ; 将 R6 的低半字与 R7 的高半字相乘，符号扩展为 32 位，将 R1:R2 相加并写入 ; R1:R2 SMLALTB R2, R1, R6, R7 ; 将 R6 的高半字与 R7 的低半字相乘，符号扩展为 32 位，将 R1:R2 相加并写入 ; R1:R2 SMLALD R6, R8, R5, R1 ; 将 R5 和 R1 的高半字相乘，以及 R5 和 R1 的低半字相乘，将 R8:R6 相加并写入 ; R8:R6 SMLALDX R6, R8, R5, R1 ; 将 R5 的高半字与 R1 的低半字相乘，以及 R5 的低半字与 R1 的高半字相乘，将 R8:R6 相加并写入 ; R8:R6

3.6.6 SMLSD 和 SMLSLD

带符号的乘减双精度运算和带符号的乘减长双精度运算

语法

操作{X}{cond} Rd, Rn, Rm, Ra

其中:

- op 是以下之一：SMLSD: 带符号乘法减法双精度。SMLSDX: 带符号乘法减法双精度反转 SMLSLD: 带符号乘法减法长双精度。SMLSLDX: 带符号乘法减法长双精度反转。 – 如果存在 X，乘法操作为底部 × 顶部和顶部 × 底部。 – 如果省略 X，乘法操作为底部 × 底部和顶部 × 顶部。

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 、‘*Rm*’是存储第一个和第二个操作数的寄存器
- ‘*Ra*’ 是保存累加值的寄存器

运营

SMLSD指令将第一个和第二个操作数的值解释为四个有符号的半字。该指令:

1. 可选地旋转第二个操作数的半字。2. 执行两个带符号的16 × 16位半字乘法运算。3. 从高半字乘法运算的结果中减去低半字乘法运算的结果。4. 将带符号的累加值加到减法运算的结果中。5. 将加法运算的结果写入目标寄存器。

SMLSLD指令将*Rn*和*Rm*的值解释为四个带符号的半字。

此指令:

1. 可选地旋转第二个操作数的半字。 2. 执行两个带符号的16位半字乘法运算，使用×。 3. 将高半字乘法运算的结果减去低半字乘法运算的结果。 4. 将*RdHi*和*RdLo*中的64位值加到减法运算的结果中。 5. 将加法运算的64位结果写入*RdHi*和*RdLo*。

限制

在这些说明中：不要使用SP或PC。

条件标志

如果累加操作溢出，此指令将设置Q标志。溢出不会在乘法运算或减法期间发生。

用于小指 在指令集，这些指令不会影响condit 离子代码标志

示例

SMLS R0, R4, R5, R6 ; 将R4的低半字与R5的低半字相乘，将R4的高半字与R5的高半字相乘，用第一个减去第二个，加上R6，写入R0 SMLSDX R1, R3, R2, R0 ; 将R3的低半字与R2的高半字相乘，将R3的高半字与R2的低半字相乘，用第一个减去第二个，加上R0，写入R1 SMLSLD R3, R6, R2, R7 ; 将R6的低半字与R2的低半字相乘，将R6的高半字与R2的高半字相乘，用第一个减去第二个，加上R6:R3，写入R6:R3 SMLSLDX R3, R6, R2, R7 ; 将R6的低半字与R2的高半字相乘，将R6的高半字与R2的低半字相乘，用第一个减去第二个，加上R6:R3，写入R6:R3.

3.6.7 SMMLA 和 SMMLS

有符号的最高有效字乘积累加和有符号的最高有效字乘法减

语法

操作{R}{cond} Rd, Rn, Rm, Ra

其中:

- op 是以下之一：SMMLA: 带符号的最高有效字乘积累加. SMMLS: 带符号的最高有效字乘法减.
- R 是一个舍入误差标志。如果指定了 R，结果将进行四舍五入而不是截断，在提取高字之前会将 0x80000000 加到乘积上。
- ‘*cond*’ 是一个可选条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’、‘*Rm*’ 是保存第一和第二乘法操作数的寄存器
- ‘*Ra*’是存储累加值的寄存器

运营

SMMLA指令将*Rn*和*Rm*中的值解释为带符号的32位字:
1. 将 *Rn* 和 *Rm* 中的值相乘。 2. 可选地通过添加 0x80000000 来对结果进行舍入。 3. 提取结果的最高有效32位。 4. 将 *Ra* 的值与带符号的提取值相加。 5. 将相加的结果写入 *Rd*。

SMMLS 指令将来自 *Rn* 和 *Rm* 的值解释为带符号的 32 位字:
1. 将 *Rn* 和 *Rm* 中的值相乘。 2. 可选地通过添加 0x80000000 对结果进行四舍五入。 3. 提取结果的最高有效32位。 4. 从 *Ra* 中的值减去提取的值。 5. 将减法结果写入 *Rd*。

限制

在这些说明中：不要使用SP或PC。

条件标志

这些指令不会影响状态标志。

示例

SMMLA R0, R4, R5, R6 ; 将 R4 和 R5 相乘，提取高 32 位， ; 加上 R6，截断并写入 R0 SMMLAR R6, R2, R1, R4 ; 将 R2 和 R1 相乘，提取高 32 位， ; 加上 R4，舍入并写入 R6 SMMLSR R3, R6, R2, R7 ; 将 R6 和 R2 相乘，提取高 32 位， ; 减去 R7，舍入并写入 R3

SMMLS R4, R5, R3, R8 ; 将 R5 和 R3 相乘，提取最高 32 位，；减去 R8，截断并将结果写入 R4。

3.6.8 SMMUL

带符号的最高有效字乘法

语法

操作{R}{cond} Rd, Rn, Rm

其中：

- op 是以下之一：SMMUL：带符号的最高字乘法。R：舍入误差标志。如果指定了 R，则结果进行舍入而非截断。在这种情况下，在提取高字之前，会将常量 0x80000000 加到乘积中。

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 和 ‘*Rm*’ 是用于存储第一个和第二个操作数的寄存器。

运营

SMMUL指令将*Rn*和*Rm*中的值解释为二进制补码的32位有符号整数。SMMUL指令:

1. 将来自 *Rn* 和 *Rm* 的值相乘。 2. 可选地对结果进行四舍五入，
否则进行截断。 3. 将结果的有符号32位最高有效位写入 *Rd*。

限制

在此指令中：不要使用SP或PC。

条件标志

该指令不会影响状态标志位。

示例

SMULL R0, R4, R5 ; 将R4和R5相乘，截断最高32位 ; 并写入R0

SMULLR R6, R2 ; 将 R6 和 R2 相乘，对最高 32 位进行四舍五入；并将结果写入 R6.

3.6.9 SMUAD 和 SMUSD

有符号双乘加和有符号双乘减

语法

操作{X}{cond} Rd, Rn, Rm

其中:

- op 是以下之一：SMUAD: 带符号双乘加. SMUADX: 带符号双乘加反转. SMUSD: 带符号双乘减. SMUSDX: 带符号双乘减反转。– 如果存在 X，乘法运算为底 × 顶和顶 × 底. 如果省略 X，乘法运算为底 × 底和顶 × 顶。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’、‘*Rm*’是存储第一个和第二个操作数的寄存器

运营

SMUAD 国际prets 第一个和第二个操作数的值作为两个带符号的ha lfwords: 1. 可选地旋转第二个操作数的半字。2. 执行两个带符号的16 × 16位乘法运算。3. 将两次乘法结果相加在一起。4. 将加法结果写入目标寄存器。

SMUSD 将第一个和第二个操作数的值解释为二进制补码有符号整数:

1. 可选地旋转第二个操作数的半字。2. 执行两个有符号的16 × 16位乘法运算。3. 将高半字乘法运算的结果从低半字乘法运算的结果中减去。4. 将减法结果写入目标寄存器。

限制

在这些说明中：不要使用SP或PC。

条件标志

如果加法溢出，则设置Q标志位。乘法不会溢出。

示例

SMUAD R0, R4, R5 ; 将 R4 的低半字与 R5 的低半字相乘，加上 R4 的高半字与 R5 的高半字相乘的结果，写入 R0 SMUADX R3, R7, R4 ; 将 R7 的低半字与 R4 的高半字相乘，加上 R7 的高半字与 R4 的低半字相乘的结果，写入 R3

SMUSD R3, R6, R2 ; 用 R4 的底部半字与 R6 的底部半字相乘，用 R6 的顶部半字与 R3 的顶部半字相乘，结果存入 R3 SMUSDX R4, R5, R3 ; 用 R5 的底部半字与 R3 的顶部半字相乘，用 R5 的顶部半字与 R3 的底部半字相乘，结果存入 R4.

3.6.10 SMUL 和 SMULW

有符号乘法（半字）和有符号乘法（字与半字）

语法

操作{XY}{cond} Rd, Rn, Rm 操
作{Y}{cond} Rd. Rn, Rm

- op 是以下之一：SMUL{XY} 带符号乘法（半字）。X 和 Y 指定源寄存器 *Rn* 和 *Rm* 中的哪个半字作为第一个和第二个乘法操作数。如果 X 是 B，则使用 *Rn* 的低半字（位 [15:0]）。 如果 X 是 T，则使用 *Rn* 的高半字（位 [31:16]）。如果 Y 是 B，则使用 *Rm* 的低半字（位 [15:0]）。 如果 Y 是 T，则使用 *Rm* 的高半字（位 [31:16]）。SMULW{Y} 带符号乘法（字乘半字）。Y 指定源寄存器 *Rm* 中的哪个半字作为第二个乘法操作数。如果 Y 是 B，则使用 *Rm* 的低半字（位 [15:0]）。 如果 Y 是 T，则使用 *Rm* 的高半字（位 [31:16]）。

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’、‘*Rm*’是存储第一个和第二个操作数的寄存器

运营

SMULBB、SMULTB、SMULBT 和 SMULTT 指令将 *Rn* 和 *Rm* 中的值解释为四个有符号的 16 位整数。这些指令：

1. 将指定的有符号半字（上或下）值从 *Rn* 和 *Rm* 中相乘。 2. 将相乘的32位结果写入 *Rd*。

SMULWT和SMULWB指令将*Rn*中的值解释为一个32位带符号整数，并将*Rm*解释为两个半字16位带符号整数。这些指令：

1. 将第一个操作数与第二个操作数的高半字（T后缀）或低半字（B后缀）相乘。 2. 将48位结果的32位有符号最高有效位写入目标寄存器。

限制

不要使用SP或PC。

示例

SMULBT R0, R4, R5 ; 将 R4 的低字半与 R5 的高字半相乘，结果写入 R0

SMULBB R0, R4, R5 ; 将R4的低半字与R5的低半字相乘，结果写入R0 SMULTT R0, R4, R5 ; 将R4的高半字与R5的高半字相乘，结果写入R0 SMULTB R0, R4, R5 ; 将R4的高半字与R5的低半字相乘，结果写入R0 SMULWT R4, R5, R3 ; 将R5与R3的高半字相乘，提取高32位并写入R4 SMULWB R4, R5, R3 ; 将R5与R3的低半字相乘，提取高32位并写入R4.

3.6.11 无符号长乘长，无符号长乘长累加，有符号长乘长，有符号长乘长累加

带符号和无符号的长整数乘法，可选累加，使用32位操作数并产生64位结果。{v*}

语法

操作{cond} RdLo, RdHi, Rn, Rm

其中:

- *op*'是以下之一： UMULL: 无符号长乘法。 UMLAL: 无符号长乘法，带累加。 SMULL: 带符号长乘法。 SMLAL: 带符号长乘法，带累加。

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- “*RdHi, RdLo*”是目标寄存器。对于UMLAL和SMLAL指令，它们还保存累积值。

- ‘*Rn*’、‘*Rm*’是保存操作数的寄存器

运营

UMULL指令将*Rn*和*Rm*中的值解释为无符号整数。它将这些整数相乘，并将结果的最低有效32位存入*RdLo*，将结果的最高有效32位存入*RdHi*。

UMLAL指令将*Rn*和*Rm*中的值视为无符号整数。它将这些整数相乘，将64位结果与存储在*RdHi*和*RdLo*中的64位无符号整数相加，并将结果写回*RdHi*和*RdLo*。

SMULL 指令将 *Rn* 和 *Rm* 中的值解释为二进制补码有符号整数。它将这些整数相乘，并将结果的低32位存入 *RdLo*，将结果的高32位存入 *RdHi*。

SMLAL指令将*Rn*和*Rm*中的值解释为二进制补码有符号整数。它将这些整数相乘，将64位结果与存储在*RdHi*和*RdLo*中的64位有符号整数相加，并将结果写回*RdHi*和*RdLo*。

限制

在这些说明中:

- 不要使用SP或PC
- *RdHi*并且 *RdLo* 必须是不同的寄存器。

条件标志

这些指令不会影响状态标志。

示例

```
UMULL R0, R4, R5, R6 ; 无符号 (R4,R0) = R5 × R6
SMLAL      R4, R5, R3, R8   ; 有符号的 (R5,R4) = (R5,R4) + R3乘以R8
```

3.6.12 SDIV 和 UDIV

带符号的除法和无符号的除法

语法

带符号除法{cond} {Rd,} Rn, R
m 无符号除法{cond} {Rd,} Rn,
Rm
其中:

- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- ‘*Rd*’ 是目标寄存器。如果省略*Rd*，则目标寄存器为*Rn.*
- ‘*Rn,*’ 是保存待除值的寄存器。
- ‘*Rm*’ 是一个保存除数的寄存器。

运营

SDIV 对 *Rn* 中的值与 *Rm* 中的值执行带符号整数除法。

UDIV对*Rn*中的值和*Rm*中的值执行无符号整数除法。

对于这两种指令，如果 *Rn* 中的值不能被 *Rm* 中的值整除，则结果将向零取整。

限制

不要使用SP或PC.

条件标志

这些指令不会更改标志。

示例

```
SDIV R0, R2, R4; 带符号除法, R0 = R2/R4 UDIV R8, R8, R1;
无符号除法, R8 = R8/R1
```

3.7 饱和指令

本节描述饱和指令。

表30. 饱和指令

Mnemonic	Brief description	See
SSAT	Signed Saturate	*SSAT and USAT on page 126*
SSAT16	Signed Saturate Halfword	*SSAT16 and USAT16 on page 127*
USAT	Unsigned Saturate	*SSAT and USAT on page 126*
USAT16	Unsigned Saturate Halfword	*SSAT16 and USAT16 on page 127*
QADD	Saturating Add	*QADD and QSUB on page 128*
QSUB	Saturating Subtract	*QADD and QSUB on page 128*
QSUB16	Saturating Subtract 16	*QADD and QSUB on page 128*
QASX	Saturating Add and Subtract with Exchange	*QASX and QSAX on page 129*
QSAX	Saturating Subtract and Add with Exchange	*QASX and QSAX on page 129*
QDADD	Saturating Double and Add	*QDADD and QDSUB on page 130*
QDSUB	Saturating Double and Subtract	*QDADD and QDSUB on page 130*
UQADD16	Unsigned Saturating Add 16	*UQADD and UQSUB on page 132*
UQADD8	Unsigned Saturating Add 8	*UQADD and UQSUB on page 132*
UQASX	Unsigned Saturating Add and Subtract with Exchange	*UQASX and UQSAX on page 131*
UQSAX	Unsigned Saturating Subtract and Add with Exchange	*UQASX and UQSAX on page 131*
UQSUB16	Unsigned Saturating Subtract 16	*UQADD and UQSUB on page 132*
UQSUB8	Unsigned Saturating Subtract 8	*UQADD and UQSUB on page 132*

对于带符号的*n*位饱和度，这意味着：

- 如果需要饱和的值小于 -2^{n-1}，则返回的结果是 -2^{n-1}
- 如果待饱和的值大于 $2^{n-1}-1$，则返回的结果是 $2^{n-1}-1$
- 否则，返回的结果与{v*}相同。

对于无符号 *n* 位饱和，这意味着：

- 如果待饱和的值小于0，则返回的结果是0
- 如果待饱和的值大于 2^{n-1}，返回的结果是 2^{n-1}
- 否则，返回的结果与{v*}相同。

如果返回的结果与饱和值不同，则称为饱和。如果发生饱和，指令会将Q标志设置为1在APSR中。否则，保持Q标志不变。要清除Q标志为0，必须使用MSR指令，参见*MSR on page 187*。

要读取Q标志的状态，请使用MRS指令，参见*MRS on page 186*。

3.7.1 SSAT和USAT

对任意位位置进行有符号饱和和无符号饱和操作，可选的移位在饱和之前进行。

语法

操作码{cond} 目标寄存器, 立即数 n, 源寄存器 {, 移位 #s}

其中:

- *op*'属于以下之一：SSAT：将带符号值饱和到带符号范围。USAT：将带符号值饱和到无符号范围。
- '*cond*' 是一个可选的条件码（参见 *Conditional execution on page 65*）。
- '*Rd*' 是目标寄存器。
- "*n*" 指定饱和到的位位置：n 范围从 1 到 32 对于 SSAT，n 范围从 0 到 31 对于 USAT。
- '*Rm*'是包含饱和值的寄存器。
- '*shift #s*' 是应用于 *Rm* 的可选偏移量，在饱和前。它必须是以下之一。如下：ASR #s: 其中s的取值范围为1到31。LSL #s: 其中s的取值范围为0到31。

运营

的 se指令饱和到有符号或无符号的n位{v*}值 卢。

SSAT指令应用指定的移位，然后饱和到有符号范围 $-2^{n-1} \le x \le 2^{n-1}-1$。

USAT指令应用指定的移位，然后饱和到无符号范围 $0 \le x \le 2^{n-1}$。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

当饱和发生时，这些指令将Q标志设置为1。

示例

SSAT R7, #16, R7, LSL #4; 逻辑左移R7中的值4位，然后; 饱和为带符号的16位值，并; 写回R7

USATNE R0, #7, R5; 有条件地将R5中的值作为无符号7位值进行饱和处理，并写入R0。

3.7.2 SSAT16 和 USAT16

有符号饱和和无符号饱和可应用于两个半字的任意位位置。

语法

操作码{cond} Rd, #n, Rm

其中:

- *op*'是以下之一：SSAT16 将有符号半字值饱和到有符号范围。USA T16 将有符号半字值饱和到无符号范围。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。
- '*n*' 指定饱和到的位位置：n 的范围从 1 到 16 用于 SSAT。n 的范围从 0 到 15 用于 U SAT。
- ‘*Rm*’ 是包含饱和值的寄存器。

运营

SSAT16指令:

1. 用由*n*中的位位置选择的要饱和的值对寄存器中的两个带符号的16位半字值进行饱和处理。2. 将结果写入目标寄存器作为两个带符号的16位半字。

usat16指令:

1. 将寄存器中的两个无符号16位半字值饱和为由*n*中的位位置选择的值。2. 将结果作为两个无符号半字写入目标寄存器。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

如果发生饱和，这些指令将Q标志设置为 1.

示例

SSAT16 R7, #9, R2 ; 饱和处理 R2 的高字和低字作为 9 位值，写入 R7 对应的半字 USAT16NE R0, #13, R5 ; 条件性地饱和处理 R5 的高字和低字作为 13 位值，写入 R0 对应的半字

3.7.3 QADD 和 QSUB

饱和加法和饱和减法，带符号的。

语法

操作{条件} Rd, Rn, Rm 操作
{条件} Rd, Rn, Rm

其中：

- *op*’是以下之一：QADD：32位饱和加法。QADD8：四次8位整数饱和加法。QADD16：两次16位整数饱和加法。QSUB：32位饱和减法。QSUB8：四次8位整数饱和减法。QSUB16：两次16位整数饱和减法。
- ‘*cond*’ 是一个可选的条件码（参见 *Conditional execution on page 65*）
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn, Rm*’ 是存储第一个和第二个操作数的寄存器。

运营

这些指令将两个、四个或八个值从第一个和第二个操作数中相加或相减，然后将一个有符号的饱和值写入目标寄存器。

QADD 和 QSUB 指令执行指定的加法或减法，然后将结果饱和到有符号范围 $-2n-1 \le x \le 2n-1-1$，其中 *x* 由指令中应用的位数决定，为 32、16 或 8。

如果返回的结果与饱和值不同，则称为饱和。如果发生饱和，QADD和QSUB指令会将APSR的Q标志设置为1。否则，Q标志保持不变。8位和16位的QADD和QSUB指令始终不会改变Q标志。

要将Q标志清除为0，必须使用MSR指令，参见 *MSR on page 187.* 。要读取Q标志的状态，请使用MRS指令，参见 *MRS on page 186*。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态码标志。 如果发生饱和，这些指令将Q标志设置为1。

示例

QADD16 R7, R4, R2 ; 将R4的半字与R2对应的半字相加，饱和到16位并写入R7对应的半字 QADD8 R3, R1, R6 ; 将R1的字节与R6对应的字节相加，饱和到8位并写入R3对应的字节 QSUB16 R4, R2, R3 ; 从R3的半字中减去对应的半字

; R2的，饱和到16位，写入R4对应的；半字QSUB8 R4, R2, R5 ; 从R2对应的字节中减去R5的字节；饱和到8位，写入R4对应的字节

3.7.4 QASX和QSAX

带交换的饱和加法和减法，带交换的饱和减法和加法，有符号的

语法

op{cond} {Rd}, Rm, Rn

其中:

- *op*'是以下之一：QASX 加减带交换和饱和。QSAX 减加带交换和饱和。
- '*cond*' 是一个可选的条件码（参见 *Conditional execution on page 65*）
- '*Rd*' 是目标寄存器。
- '*Rn, Rm*' 是存储第一个和第二个操作数的寄存器。

运营

QASX指令:

1. 将源操作数的高字与第二操作数的低字相加。2. 从第一操作数的低字中减去第二操作数的高字。3. 饱和处理减法结果，并将一个16位有符号整数写入目标寄存器的低字，范围为–2^15 ≤ x ≤ 2^15 – 1，其中 *x* 等于 16。4. 饱和处理加法结果，并将一个16位有符号整数写入目标寄存器的高字，范围为–2^15 ≤ x ≤ 2^15 – 1，其中 *x* 等于 16。

QSAX指令:

1. 将第二个操作数的低半字从第一个操作数的高半字中减去。 2. 将源操作数的低半字与第二个操作数的高半字相加。 3. 对求和结果进行饱和处理，并将一个16位有符号整数写入目标寄存器的低半字，该整数的范围为 –215 ≤ x ≤ 215 – 1，其中 *x* 等于 16。 4. 对减法结果进行饱和处理，并将一个16位有符号整数写入目标寄存器的高半字，该整数的范围为 –215 ≤ x ≤ 215 – 1，其中 *x* 等于 16。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

示例

QASX R7, R4, R2 ; 将R4的高半字加到R2的低半字， ; 饱和运算到16位，写入到R7的高半字中

; 将R2的高字部分从R4的低半字部分减去，饱和到16位并写入R7的低半字部分

QSAX R0, R3, R5 ; 从R3的高半字中减去R5的低半字，饱和到16位，写入R0的高半字；将R3的低半字加到R5的高半字，饱和到16位，写入R0的低半字。

3.7.5 加法和减法

饱和双倍加法和饱和双倍减法，带符号的

语法

op{cond} {Rd}, Rm, Rn

其中：

- *op*'是以下之一：QDADD 饱和双倍并加。
  QDSUB 饱和双倍并减。

- '*cond*' 是一个可选的条件码（参见 *Conditional execution on page 65*）
- '*Rd*' 是目标寄存器。
- '*Rn, Rm*' 是存储第一个和第二个操作数的寄存器。

运营

QDADD指令：
1. 将第二个操作数的值加倍。2. 将加倍后的结果与第一个操作数中的带符号的饱和值相加。3. 将结果写入目标寄存器。

QDSUB指令：
1. 将第二个操作数的值加倍。 2. 将加倍后的值从第一个操作数的带符号饱和值中减去。 3. 将结果写入目标寄存器。

无论是双倍操作还是加法或减法，其结果都会被限制在32位有符号整数范围内$-231 \le x \le 231-1$。如果在任一操作中发生溢出，则会在APSR中设置Q标志。

限制

请勿使用SP和PC。

条件标志

当饱和发生时，这些指令将Q标志设置为1。

示例

QDADD R7, R4, R2 ; 将R4双精度扩展并饱和到32位，加上R2， ； 饱和到32位，写入R7 QDSUB R0, R3, R5 ; 从R3双精度扩展并饱和到32位的值中减去

; 从 R5，饱和到 32 位，写入 R0。

3.7.6 UQASX 和 UQSAX

带交换的饱和加法和减法、带交换的饱和减法和加法，无符号。

语法

op{cond} {Rd}, Rm, Rn

其中：

- *op*'是以下之一：UQASX 加减运算并饱和。UQSAX 减加运算并饱和。
- '*cond*' 是一个可选的条件码（参见 *Conditional execution on page 65*）
- '*Rd*' 是目标寄存器。
- '*Rn, Rm*' 是存储第一个和第二个操作数的寄存器。

运营

UQASX指令：

1. 将源操作数的低半字与第二个操作数的高半字相加。 2. 从第一个操作数的高半字中减去第二个操作数的低半字。 3. 对加法结果进行饱和处理，并将一个16位无符号整数（范围为$0 \le x \le 216 - 1$，其中 *x* 等于16）写入目标寄存器的高半字。 4. 对减法结果进行饱和处理，并将一个16位无符号整数（范围为$0 \le x \le 216 - 1$，其中 *x* 等于16）写入目标寄存器的低半字。

UQSAX指令：

1. 将第二个操作数的低半字从第一个操作数的高半字中减去。2. 将第一个操作数的低半字与第二个操作数的高半字相加。3. 对减法结果进行饱和处理，并将一个16位无符号整数（范围为$0 \le x \le 216 - 1$，其中*x*等于16）写入目标寄存器的高半字。4. 对加法结果进行饱和处理，并将一个16位无符号整数（范围为$0 \le x \le 216 - 1$，其中*x*等于16）写入目标寄存器的低半字。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

示例

UQASX R7, R4, R2 ; 将R4的高字半与R2的低字半相加， ; 饱和到16位，写入R7的高字半 ; 从...的低字半中减去R2的高字半

; R4，饱和到16位，写入R7的低半字 UQSAX R0, R3, R5 ; 从R3的高半字中减去R5的低半字，饱和到16位，写入R0的高半字 ; 将R4的低半字加到R5的高半字上；饱和到16位，写入R0的低半字.

3.7.7 UQADD和UQSUB

无符号的饱和加法和饱和减法

语法

操作{条件} Rd, Rn, Rm 操作
{条件} Rd, Rn, Rm

其中:

- *op*'是以下之一：UQADD8 饱和四次无符号8位整数加法。UQADD16 饱和两次无符号16位整数加法。UDSUB8 饱和四次无符号8位整数减法。UQSUB16 饱和两次无符号16位整数减法。

- '*cond*' 是一个可选的条件码（参见 *Conditional execution on page 65*）
- '*Rd*' 是目标寄存器。
- '*Rn, Rm*' 是存储第一个和第二个操作数的寄存器。

运营

这些指令对两个或四个值进行加法或减法运算，然后将无符号饱和值写入目标寄存器。

UQADD16指令:
1. 将第一和第二操作数的高半字和低半字分别相加。2. 对目标寄存器中每个半字的加法结果进行饱和处理，使其处于无符号范围 0 ≤ x ≤ 216-1，其中 *x* 为 16。

UQADD8 指令:
1. 将第一个操作数和第二个操作数的每个对应字节相加。2. 对目标寄存器中的每个字节的加法结果进行饱和处理，使其处于无符号范围 0 ≤ x ≤ 28-1，其中 *x* 是 8。

UQSUB16指令:
1. 从第一个操作数的相应半字中减去第二个操作数的两个半字。2. 将目标寄存器中差值的结果饱和到无符号范围

0 ≤ x ≤ 216-1，其中 *x* 是 16。

UQSUB8 指令:
1. 减去第二个操作数的对应字节，从的对应字节
第一个操作数。2。对目标寄存器中每个字节的差值结果进行饱和处理，使其处于无符号范围 0 ≤ x ≤ 28-1，其中 *x* 是 8。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响状态标志。

示例

UQADD16 R7, R4, R2; 将 R4 中的半字加到 R2 中的对应半字上， ; 饱和到 16 位，写入 R7 的对应半字 ; UQADD8 R4, R2, R5 ; 将 R2 的字节加到 R5 的对应字节上，饱和 ; 到 8 位，写入 R4 的对应字节 ; UQSUB16 R6, R3, R0 ; 将 R0 中的半字从 R3 中的对应半字中减去，饱和到 16 位，写入 ; R6 的对应半字 ; UQSUB8 R1, R5, R6 ; 将 R6 中的字节从 R5 的对应字节中减去， ; 饱和到 8 位，写入 R1 的对应字节。

3.8 包装和拆包说明

*Table 31*显示了操作数据打包和解包的指令:

表31. 包装和拆卸说明

Mnemonic	Brief description	See
PKH	Pack Halfword	*PKHBT and PKHTB on page 135*
SXTAB	Extend 8 bits to 32 and add	*SXTA and UXTA on page 137*
SXTAB16	Dual extend 8 bits to 16 and add	*SXTA and UXTA on page 137*
SXTAH	Extend 16 bits to 32 and add	*SXTA and UXTA on page 137*
SXTB	Sign extend a byte	*SXT and UXT on page 136*
SXTB16	Dual extend 8 bits to 16 and add	*SXT and UXT on page 136*
SXTH	Sign extend a halfword	*SXT and UXT on page 136*
UXTAB	Extend 8 bits to 32 and add	*SXTA and UXTA on page 137*
UXTAB16	Dual extend 8 bits to 16 and add	*SXTA and UXTA on page 137*
UXTAH	Extend 16 bits to 32 and add	*SXTA and UXTA on page 137*
UXTB	Zero extend a byte	*SXT and UXT on page 136*
UXTB16	Dual zero extend 8 bits to 16 and add	*SXT and UXT on page 136*
UXTH	Zero extend a halfword	*SXT and UXT on page 136*

3.8.1 PKHBT 和 PKHTB

打包半字

语法

操作{cond} {Rd}, Rn, Rm {, 逻辑左移 #imm} 操
作{cond} {Rd}, Rn, Rm {, 算术右移 #imm}

其中:

- *op*'是以下之一：PKHBT 打包半字，底部和顶部带移位。PKHTB 打包半字，顶部和底部带移位。

- '*cond*' 是一个可选的条件码（参见 *Conditional execution on page 65*）
- '*Rd*' 是目标寄存器。
- '*Rn*' 是第一个操作数寄存器。
- '*Rm*' 是第二个操作数寄存器，保存的值可被选移位。
- '*imm*' 是移位长度。移位长度的类型取决于指令：对于 PKHBT：LSL：从 1 到 31 的逻辑左移，0 表示无移位。对于 PKHTB：ASR：从 1 到 32 的算术右移，32 位移位编码为 0b00000。

运营

PKHBT指令:
1. 将第一个操作数的低半字值写入目标寄存器的低半字部分。2. 如果移位，则将第二个操作数的移位后值写入目标寄存器的高半字部分。

PKHTB 指令:
1. 将第一个操作数的高半字写入目标寄存器的高半字。2. 若移位，将第二个操作数的移位值写入目标寄存器的低半字。

限制

Rd不得为SP且不得为PC。

条件标志

该指令不会改变标志。

示例

PKHBT R3, R4, R5 LSL #0 ; 将R4的低半字写入R3的低半字，将R5的高半字（未移位）写入R3的高半字 PKHTB R4, R0, R2 ASR #1 ; 将R2算术右移1位后的内容写入R4的低半字，并将R0的高半字写入R4的高半字。

3.8.2 SXT 和 UXT

符号扩展和零扩展。

语法

操作码{条件} {Rd,} Rm {, ROR #n} 操
作码{条件} {Rd}, Rm {, ROR #n}

其中:

- *op*'是以下之一：SXTB 符号扩展一个8位值到32位值。 SXTH 符号扩展一个16位值到32位值。 SXTB16 符号扩展两个8位值到两个16位值。 UXTB 零扩展一个8位值到32位值。 UXTH 零扩展一个16位值到32位值。 UXTB16 零扩展两个8位值到两个16位值。

- '*cond*' 是一个可选的条件码（参见 *Conditional execution on page 65*）
- '*Rd*' 是目标寄存器。
- "*Rm*"是保存扩展值的寄存器。
- "*ROR #n*" 是其中之一：ROR #8 从 Rm 取出的值右旋转 8 位。 ROR #16 从 Rm 取出的值右旋转 16 位。 ROR #24 从 Rm 取出的值右旋转 24 位。
  如果省略 ROR #n，则不执行旋转操作。

运营

这些说明执行以下操作:

1. 将Rm中的值右移0、8、16或24位。 2. 从结果值中提取位：SXTB提取bits[7:0]，并符号扩展到32位。 UXTB提取bits[7:0]，并零扩展到32位。 SXTH提取bits[15:0]，并符号扩展到32位。 UXTH提取bits[15:0]，并零扩展到32位。 SXTB16提取bits[7:0]，并符号扩展到16位， 并提取bits[23:16]，符号扩展到16位。 UXTB16提取bits[7:0]，并零扩展到16位，

并提取位 [23:16]，零扩展到16位。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响标志。

示例

SXTH R4, R6, ROR #16 ; 将R6右移16位，获取结果的低半字 ；符号扩展为32位并写入R4 UXTB R3, R10 ; 提取R10中值的最低字节，零扩展，然后 写入R3.

3.8.3 SXTA 和 UXTA

带符号和无符号扩展与加法

语法

操作{条件} {Rd,} Rn, Rm {, ROR #n} 操作{
条件} {Rd,} Rn, Rm {, ROR #n}

其中:

- *op*'是以下之一：SXTAB 符号扩展一个8位值到32位值并加法. SXTAH 符号扩展一个16位值到32位值并加法. SXTAB16 符号扩展两个8位值到两个16位值并加法. UXTAB 零扩展一个8位值到32位值并加法. UXTAH 零扩展一个16位值到32位值并加法. UXTAB16 零扩展两个8位值到两个16位值并加法.

- '*cond*' 是一个可选的条件码（参见 *Conditional execution on page 65*）
- "*Rd*" 是目标寄存器。
- '*Rn*' 是第一个操作数寄存器。
- '*Rm*' 是保存要旋转和扩展的值的寄存器。
- '*ROR #n*' 是以下之一：ROR #8 从 Rm 获取的值右移 8 位。 ROR #16 从 Rm 获取的值右移 16 位。 ROR #24 从 Rm 获取的值右移 24 位。 如果省略 ROR #n，则不执行旋转操作。

运营

这些说明执行以下操作:

1. 将 Rm 中的值右移 0、8、16 或 24 位。 2. 从结果值中提取位：SXTAB 从 Rm 中提取 bits[7:0] 并符号扩展到 32 位。 UXTAB 从 Rm 中提取 bits[7:0] 并零扩展到 32 位。 SXTAH 从 Rm 中提取 bits[15:0] 并符号扩展到 32 位。 UXTAH 从 Rm 中提取 bits[15:0] 并零扩展到 32 位。SXTAB16 从 Rm 中提取 bits[7:0] 并符号扩展到 16 位， 并提取 bits[23:16] 从 Rm 并符号扩展到 16 位。UXTAB16 从 Rm 中提取 bits[7:0] 并零扩展到 16 位， 并提取 bits[23:16] 从 Rm 并零扩展到 16 位。3. 将带符号或零扩展的值与 Rn 的字或对应的半字相加，并将结果写入 Rd。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响标志。

示例

SXTAH R4, R8, R6, ROR #16 ; 将R6右移16位，获取低半字，符号扩展为32位，加上R8，并 ; 写入R4 UXTAB R3, R4, R10 ; 提取R10的低字节并零扩展为32位，加上R4，并写入R3.

3.9 位字段指令

*Table 32*显示在寄存器或位字段中操作相邻位组的指令。

表 32. 对相邻位集合进行操作的指令

Mnemonic	Brief description	See
BFC	Bit field clear	*BFC and BFI on page 139*
BFI	Bit field insert	*BFC and BFI on page 139*
SBFX	Signed bit field extract	*SBFX and UBFX on page 140*
SXTB	Sign extend a byte	*SXT and UXT on page 141*
SXTH	Sign extend a halfword	*SXT and UXT on page 141*
UBFX	Unsigned bit field extract	*SBFX and UBFX on page 140*
UXTB	Zero extend a byte	*SXT and UXT on page 141*
UXTH	Zero extend a halfword	*SXT and UXT on page 141*

3.9.1 BFC 和 BFI

位字段清除和位字段插入

语法

BFC{cond} Rd, #lsb, #width BFI{cond} Rd,
Rn, #lsb, #width

其中:

- ‘*cond*’ 是一个可选的条件码，参见 *Conditional execution on page 65*。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是源寄存器。
- *‘lsb’*是位字段的最低有效位的位置。*lsb* 必须位于0到31的范围内。
- *‘width’*是位字段的宽度，且必须位于1到32-*lsb*的范围内。

运营

BFC 清除寄存器中的位字段。它从低位位置 *lsb* 开始清除 *Rd* 中的宽度位。其他 *Rd* 中的位保持不变。

BFI 将一个寄存器中的位字段复制到另一个寄存器中。它将 *Rd* 中从低位位置 *lsb* 开始的宽度位，替换为 *Rn* 中从 bit[0] 开始的宽度位。 *Rd* 中的其他位保持不变。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响标志。

示例

```
BFC R4, #8, #12 ; 清除R4的位8到位19（12位）为0
BFI R9, R2, #8, #12 ; 将R9的第8位到第19位（共12位）替换为
; 位 0 到 位 11 来自 R2
```

3.9.2 带符号位字段提取和无符号位字段提取

带符号位字段提取和无符号位字段提取。

语法

SBFX{cond} Rd, Rn, #lsb, #width UBFX{co
nd} Rd, Rn, #lsb, #width

其中:

- ‘*cond*’ 是一个可选的条件码，参见 *Conditional execution on page 65*。
- ‘*Rd*’ 是目标寄存器。
- ‘*Rn*’ 是源寄存器。
- *‘lsb’*是位字段的最低有效位的位置。*lsb* 必须位于0到31的范围内。
- *‘width’*是位字段的宽度，且必须位于1到32-*lsb*的范围内。

运营

SBFX 从一个寄存器中提取位字段，将其符号扩展到32位，并将结果写入目标寄存器。

UBFX 从一个寄存器中提取位字段，对其进行零扩展到 32 位，并将结果写入目标寄存器。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响标志。

示例

SBFX R0, R1, #20, #4 ; 从 R1 中提取位 20 到位 23（4 位），符号扩展 到 32 位，然后将结果写入 R0。
UBFX R8, R11, #9, #10; 从 R11 中提取位 9 到位 18（10 位），零扩展 。

; 扩展为32位并将结果写入R8

3.9.3 SXT 和 UXT

符号扩展和零扩展。

语法

扩展SX文本{cond} {Rd,} Rm {, ROR #n} 扩展U
X文本{cond} {Rd}, Rm {, ROR #n}

其中:

- “extend” 是其中之一：B: 将8位值扩展为32位值。H: 将16位值扩展为32位值。
- "*cond*" 是一个可选的条件码，参见 *Conditional execution on page 65*。
- ‘*Rd*’ 是目标寄存器。
- *‘Rm’* 是保存用于扩展的值的寄存器。{v*}
- ROR #*n* 是以下之一： ROR #8：来自 *Rm* 的值右旋转 8 位。 ROR #16：来自 *Rm* 的值右旋转 16 位。 ROR #24：来自 *Rm* 的值右旋转 24 位。 如果省略 ROR #*n*，则不执行旋转。

运营

这些说明执行以下操作:

1. 将 *Rm* 的值循环右移 0、8、16 或 24 位。 2
从结果值中提取位：– SXTB 提取 bits[7:0] 并进行符号扩展至 32 位。– UXTB 提取 bits[7:0] 并进行零扩展至 32 位。– SXTH 提取 bits[15:0] 并进行符号扩展至 32 位。– UXTH 提取 bits[15:0] 并进行零扩展至 32 位。

限制

请勿使用SP和PC。

条件标志

这些指令不会影响标志。

示例

```
带符号扩展半字 R4, R6, 右移 #16; 将R6右移16位，然后获取结果的低 字半部分，接着符号扩展到 32位
                                  ，并将结果写入R4.
UXTB R3, R10                      ; 提取R10中值的最低字节并将其零扩展 ; 将结果写入R3
```

3.9.4 分支和控制指令

*Table 33*显示分支和控制指令:

表 33. 分支和控制指令

Mnemonic	Brief description	See
B	Branch	*B, BL, BX, and BLX on page 142*
BL	Branch with Link	*B, BL, BX, and BLX on page 142*
BLX	Branch indirect with Link	*B, BL, BX, and BLX on page 142*
BX	Branch indirect	*B, BL, BX, and BLX on page 142*
CBNZ	Compare and Branch if Non Zero	*CBZ and CBNZ on page 144*
CBZ	Compare and Branch if Non Zero	*CBZ and CBNZ on page 144*
IT	If-Then	*IT on page 145*
TBB	Table Branch Byte	*TBB and TBH on page 147*
TBH	Table Branch Halfword	*TBB and TBH on page 147*

3.9.5 B、BL、BX 和 BLX

分支指令。

语法

B{cond} 标签 BL{cond} 标签 BX{cond} Rm BLX{cond} Rm

其中:

- ‘B’ 是分支（立即数）。
- “BL” 是带链接的分支（立即数）。
- ‘BX’ 是分支间接（寄存器）。
- ‘BLX’ 是间接跳转并链接（寄存器）的指令。
- *‘cond’* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *‘label’*是PC相对的表达式。参见 *PC-relative expressions on page 65*。
- *‘Rm’* 这是一个指示分支地址的寄存器。变量 *Rm* 中的位[0] 必须为 1，但分支地址是通过将位[0] 改为 0 来生成的。

运营

这些指令都会导致分支到 *label*，或者跳转到 *Rm* 所指示的地址。此外:

- BL 和 BLX 指令将下一条指令的地址写入 LR（链接寄存器，R14）。
- 如果 *Rm* 的 bit[0] 位为 0，则 BX 和 BLX 指令会引发 UsageFault 异常。

B *cond label* 是唯一可以既在 IT 块内又在 IT 块外的条件指令。所有其他分支指令必须在 IT 块内作为条件指令，且必须在 IT 块外作为无条件指令，参见 *IT on page 145*。

*Table 34*显示各种分支指令的范围。

表34. 分支范围

Instruction	Branch range
B label	–16 MB to +16 MB
B*cond* label (outside IT block)	–1 MB to +1 MB
B*cond* label (inside IT block)	–16 MB to +16 MB
BL{*cond*} label	–16 MB to +16 MB
BX{*cond*} Rm	Any value in register
BLX{*cond*} Rm	Any value in register

您可能需要使用.W后缀以获得最大分支范围。参见 *Instruction width selection on page 68*。

限制

限制条件如下:

- 不要在BLX指令中使用PC
- 对于BX和BLX，*Rm* 的bit[0]必须为1以确保正确执行，但会发生分支到通过将bit[0]改为0创建的目标地址。
- 当这些指令中的任何一个位于IT块内时，它必须是该IT块的最后一条指令。

B*cond* 是唯一不需要位于 IT 块内的条件指令。然而，当它位于 IT 块内时，其分支范围更长。

条件标志

这些指令不会更改标志。

示例

```
B    loopA ; 跳转到 loopA BLE   ng   ; 小于或等于则跳转到标签 ng B.W   target ; 在16MB范围内跳转到 target BEQ   target ; 等于则跳转到 target BEQ.W target ; 在1MB范围内条件跳转到 target BL funC  ; 带链接的跳转（调用）到函数 funC，返回地址          ; 存储在LR BX   LR   ; 从函数调用返回 BXNE  R0   ; 如果非零则跳转到R0存储的地址
```

```
BLX R0 {v*}        ; 带链接和交换的跳转（调用）到存储的地址
                   ; 在 R0
```

3.9.6 CBZ和CBNZ

公司配对并分支为零，比较并分支为否 n-零

语法

CBZ Rn，标签 CB
NZ Rn，标签

其中：

- *'Rn'*是保存操作数的寄存器。
- *'label'*是分支目标。

运营

使用CBZ或CBNZ指令以避免修改条件码标志并减少指令数量。

CBZ Rn, 标签不会改变条件标志，但其他方面等同于：

```
比较 Rn, #0
BEQ 标签
```

CBNZ Rn, label 不会改变条件标志位，但其他方面等效于： CMP Rn, #0 B NE label

限制

限制条件如下：

- *Rn*必须位于R0到R7范围内。
- 分支目标必须位于指令之后的4到130字节范围内。
- 这些指令不得在IT块内使用。

条件标志

这些指令不会更改标志。

示例

CBZ R5, 目标 ; 向前跳转若R5为零 CBNZ R0, 目标 ; 向前跳转若R0不为零

3.9.7 信息技术

如果-则条件指令。

语法

IT{x{y{z}}} 条件

其中:

- *'x'*指定IT块中第二条指令的条件切换。
- *'y'*指定IT块中第三条指令的条件转移。
- *'z'*指定IT块中第四条指令的条件切换。
- *'cond'*规定了IT块中第一条指令的条件。

IT块中第二、第三和第四条指令的条件切换可以是以下任一种:

T: 然后。将条件 *cond* 应用于指令。E: 否则。将 *cond* 的逆条件应用于指令。

a) 可以使用AL（条件*always*）用于*cond*的IT指令中。如果这样做，IT块中的所有指令必须是无条件的，且每个*x*、*y*和*z*必须是T或省略，但不能是E。

运营

IT指令可将最多四个后续指令设为条件指令。这些条件可以全部相同，或者其中一些可以是其他条件的逻辑反面。IT指令之后的条件指令构成*IT block*。

IT块中的指令，包括任何分支，必须在它们的语法的*{cond}* 部分中指定条件。

您的汇编器可能能够自动为您生成所需的条件指令的IT指令，因此您无需自己编写它们。请参阅您的汇编器文档以获取详细信息。

位于IT块中的BKPT指令始终会被执行，即使其条件失败。

异常可以在一条指令和对应的指令块之间发生，或在指令块内部发生。这种异常会导致进入相应的异常处理程序，并在LR和堆栈中的PSR中保存适当的返回信息。

用于异常返回的指令可以正常用于从异常返回，且IT块的执行能够正确恢复。这是唯一允许PC修改指令跳转到IT块中指令的方式。

限制

以下指令在IT块中不允许使用:

- 信息技术
- CBZ 和 CBNZ
- CPSID 和 CPSIE

使用IT块时的其他限制有：

- 任何修改程序计数器（PC）的分支指令或操作，必须位于指令块之外，或必须是指令块中的最后一条指令。这些包括：– ADD PC, PC, Rm – MOV PC, Rm – B, BL, BX, BLX – 任何写入程序计数器（PC）的LDM、LDR或POP指令 – TBB和TBH
- 不要跳转到任何IT块内的指令，除非从异常处理程序返回。
- 所有条件指令（除B*cond*外）必须位于IT块内。B*cond*既可以位于IT块外，也可以位于IT块内，但若位于其中则具有更大的分支范围。
- 每个位于IT块中的指令必须指定一个条件码后缀，该后缀与其他指令中的条件码后缀相同或逻辑反。

你的汇编器可能会对IT块的使用施加额外的限制，例如禁止在其中使用汇编指令。

条件标志

该指令不会改变标志。

示例

```
ITTE NE ; 接下来的3条指令是条件性的
ANDNE R0, R0, R1 ; ANDNE 不更新条件标志
ADDSNE R2, R2, #1  ; ADDSNE 更新条件标志
MOVEQ R2, R3      ; 条件移动

比较 R0 和 #9 ; 将 R0 的十六进制值（0 到 15）转换为 ASCII
; （'0'-'9'，'A'-'F'）
ITE   GT          ; 下面2条指令是条件性的
ADDGT R1, R0, #55 ; 转换 0xA -> 'A'
ADDLE R1, R0, #48  ; 转换 0x0 -> '0'

IT GT ; IT块仅包含一个条件指令
ADDGT R1, R1, #1 ; 有条件地递增 R1

ITTEE EQ ; 接下来的4条指令是条件性的
条件移动 R0, R1      ; 条件移动
ADDEQ R2, R2, #10  ; 条件加
ANDNE R3, R3, #1   ; 条件性 AND
BNE.W  dloop       ; 分支指令只能用于最后
; IT模块的指令

IT NE ; 下一条指令是条件指令
ADD R0, R0, R1 ; 语法错误：IT块中未使用条件码
```

3.9.8 TBB和TBH

分支字节表 和 分支半字表。

语法

TBB [Rn, Rm] TBH [Rn, Rm
, LSL #1]

其中:

- *'Rn'*这是包含分支长度表地址的寄存器。如果 *Rn* 是 PC，则表的地址是紧跟在 TB B 或 TBH 指令之后的字节的地址。
- *'Rm'* 是索引寄存器。该寄存器包含一个指向表的索引。对于半字表，LSL #1将{v*} 中的值加倍，以形成表中的正确偏移量。

运营

这些指令使用一个单字节偏移量表进行PC相对向前分支（TBB），或使用半字偏移量表（TB H）。*Rn*提供一个指向该表的指针，*Rm*提供一个表中的索引。对于TBB，分支偏移量是表中返回的字节无符号值的两倍；对于TBH，分支偏移量是表中返回的半字无符号值的两倍。分支将跳转到TBB或TBH指令后立即的字节地址处的偏移量地址。

限制

限制条件如下:

- *Rn*不得是 SP
- *Rm*不得为 SP 且不得为 PC
- 当这些指令中的任何一条在IT块内使用时，它必须是IT块的最后一条指令。

条件标志

这些指令不会更改标志。

示例

```
ADR.W R0, BranchTable_Byte  TBB [R0, R1] ; R1 是索引，R0 是分支表的基地址  Case1 ；指令序列如下
 Case2 ；指令序列如下  Case3 ；指令序列如下  BranchTable_Byte     DCB   0              ; Case1 偏移量计
算     DCB   ((Case2-Case1)/2) ; Case2 偏移量计算     DCB   ((Case3-Case1)/2) ; Case3 偏移量计算     T
BH   [PC, R1, LSL #1]  ; PC 是分支表的基地址，R1 是索引                          ; 分支表
```

```
BranchTable_H    DCI    ((CaseA - BranchTable_H)/2) ; CaseA偏移量计算    DCI    ((CaseB -
BranchTable_H)/2) ; CaseB偏移量计算    DCI    ((CaseC - BranchTable_H)/2) ; CaseC偏移量
计算  CaseA ；以下为指令序列  CaseB ；以下为指令序列  CaseC ；以下为指令序列
```

3.10 浮点指令

这些指令仅在系统中包含并启用了FPU时才可用。参见 *Enabling the FPU on page 257* 以获取关于启用浮点运算单元的信息。

表35. 浮点指令

Mnemonic	Brief description	See
VABS	Floating-point Absolute	*VABS on page 151*
VADD	Floating-point Add	*VADD on page 152*
VCMP	Compare two floating-point registers, or one floating-point register and zero	*VCMP, VCMPE on page 153*
VCMPE	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	*VCMP, VCMPE on page 153*
VCVT	Convert between floating-point and integer	*VCVT, VCVTR between floating-point and integer on page 154*
VCVT	Convert between floating-point and fixed point	*VCVT between floating-point and fixed-point on page 155*
VCVTR	Convert between floating-point and integer with rounding	*VCVT, VCVTR between floating-point and integer on page 154*
VCVTB	Converts half-precision value to single-precision	*VCVTB, VCVTT on page 156*
VCVTT	Converts single-precision register to half-precision	*VCVTB, VCVTT on page 156*
VDIV	Floating-point Divide	*VDIV on page 157*
VFMA	Floating-point Fused Multiply Accumulate	*VFMA, VFMS on page 158*
VFNMA	Floating-point Fused Negate Multiply Accumulate	*VFNMA, VFNMS on page 159*
VFMS	Floating-point Fused Multiply Subtract	*VFMA, VFMS on page 158*
VFNMS	Floating-point Fused Negate Multiply Subtract	*VFNMA, VFNMS on page 159*
VLDM	Load Multiple extension registers	*VLDM on page 160*
VLDR	Loads an extension register from memory	*VLDR on page 161*
VLMA	Floating-point Multiply Accumulate	*VLMA, VLMS on page 162*
VLMS	Floating-point Multiply Subtract	*VLMA, VLMS on page 162*
VMOV	Floating-point Move Immediate	*VMOV immediate on page 163*
VMOV	Floating-point Move Register	*VMOV register on page 164*
VMOV	Copy Arm core register to single precision	*VMOV scalar to Arm core register on page 165*
VMOV	Copy 2 Arm core registers to 2 single precision	*VMOV Arm core register to single precision on page 166*
VMOV	Copies between Arm core register to scalar	*VMOV two Arm core registers to two single precision on page 167*
VMOV	Copies between Scalar to Arm core register	*VMOV Arm Core register to scalar on page 168*
VMRS	Move to Arm core register from floating-point System Register	*VMRS on page 169*

表35. 浮点指令

Mnemonic	Brief description	See
VMSR	Move to floating-point System Register from Arm Core register	*VMSR on page 170*
VMUL	Multiply floating-point	*VMUL on page 171*
VNEG	Floating-point negate	*VNEG on page 172*
VNMLA	Floating-point multiply and add	*VNMLA, VNMLS, VNMUL on page 173*
VNMLS	Floating-point multiply and subtract	*VNMLA, VNMLS, VNMUL on page 173*
VNMUL	Floating-point multiply	*VNMLA, VNMLS, VNMUL on page 173*
VPOP	Pop extension registers	*VPOP on page 174*
VPUSH	Push extension registers	*VPUSH on page 175*
VSQRT	Floating-point square root	*VSQRT on page 176*
VSTM	Store Multiple extension registers	*VSTM on page 177*
VSTR	Stores an extension register to memory	*VSTR on page 178*
VSUB	Floating-point Subtract	*VSUB on page 179*

3.10.1 VABS

浮点绝对。

语法

VABS{cond}.F32 Sd, Sm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd, Sm'*是目标浮点值和操作数浮点值。

运营

此指令:

1. 取操作数浮点寄存器的绝对值。2. 将结果存入目标浮点寄存器。

限制

没有限制。

条件标志

浮点指令清除符号位。

示例

VABS.F32 S4, S6

3.10.2 VADD

浮点加法

语法

VADD{cond}.F32 {Sd,} Sn, Sm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标浮点值
- *'Sn, Sm'*操作数的浮点值。

运营

此指令:
1. 将两个浮点操作数寄存器中的值相加。2. 将结果放入目标浮点寄存器。

限制

没有限制。

条件标志

该指令不会改变标志。

示例

向量加法指令 VADD.F32，操作数为 S4, S6, S7

3.10.3 VCMP, VCMPE

比较两个浮点寄存器，或一个浮点寄存器和零。{v*}

语法

VCMP{E}{cond}.F32 Sd, Sm VCMP
{E}{cond}.F32 Sd, #0.0

其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'E'*如果存在，任何NaN操作数都会引发无效操作异常。否则，只有信号NaN会引发该异常。
- *'Sd'*是用于比较的浮点操作数。
- *'Sm'*是与{v*}进行比较的浮点操作数

运营

此指令：

1. 比较：两个浮点寄存器。一个浮点寄存器与零。 1. 将结果写入FPSCR标志。

限制

如果任一操作数是任何类型的非数，此指令可能会引发无效操作异常。如果任一操作数是信号非数，此指令始终会引发无效操作异常。

条件标志

当此指令将结果写入FPSCR标志时，这些值通常通过后续的VMRS指令转移到Arm标志，参见*VMRS on page 169*。

示例

VCMP.F32 S4, #0.0 VCMP
.F32 S4, S2

三.十.四 VCVT、VCVTR 在浮点数和内之间 整数

将寄存器中的值从浮点数转换为32位整数。{v*}

语法

VCVT{R}{cond}.Tm.32位浮点数 Sd,
Sm VCVT{cond}.32位浮点数.Tm Sd,
Sm
其中:

- *'R'* 如果指定了 R，该操作使用由 FPSCR 指定的舍入模式。如果省略 R。该操作使用向零舍入的舍入模式。
- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Tm'* 是操作数的数据类型。它必须是以下之一：S32 有符号 32 位值。U32 无符号 32 位值。
- *'Sd, Sm'* 是目标寄存器和操作数寄存器。

运营

这些说明:
1. 或者
- 将寄存器中的值从浮点数转换为32位整数。{v*}
- 将32位整数转换为浮点值。2. 将结果存入第二个寄存器。

浮点数到整数的转换通常使用向零舍入模式，但可以可选地使用由FPSCR指定的舍入模式。

整数到浮点数的转换操作使用由FPSCR指定的舍入模式。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.5 在浮点和定点之间的VCVT 整数

将寄存器中的值在浮点数和定点数之间进行转换。{v*}

语法

VCVT{cond}.向双精度.F32 Sd, Sd, #fbits VC
VT{cond}.F32.向双精度 Sd, Sd, #fbits

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Td'*这是定点数的数据类型。它必须是以下之一：S16 带符号的16位值。U16 无符号的16位值。S32 带符号的32位值。U32 无符号的32位值。

- *'Sd'*是目标寄存器和操作数寄存器。
- *'fbits'*是定点数中的分数位数：如果 Td 是 S16 或 U16，则 fbits 必须在 0-16 的范围内。如果 Td 是 S32 或 U32，则 fbits 必须在 1-32 的范围内。

运营

这些说明:

或者

将寄存器中的浮点值转换为定点值。

将寄存器中的值从定点数转换为浮点数。

将结果存入第二个寄存器。

浮点数是单精度的。{v*}

定点值可以是16位或32位。转换从定点值获取操作数时，会从源寄存器的低阶位读取，并忽略任何剩余位。

有符号转换到定点值时，会将结果值的符号位扩展到目标寄存器的宽度。

无符号转换到定点值时，将结果值零扩展到目标寄存器宽度。

浮点到定点的运算使用向零舍入的舍入模式。定点到浮点的运算使用四舍五入的舍入模式。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.6 VCVTB, VCVTT

在半精度值和单精度值之间进行转换。

语法

VCVT{y}{cond}.F32.F16 Sd, Sm VCVT
{y}{cond}.F16.F32 Sd, Sm

其中:

- *'y'*指定操作数寄存器 Sm 或目标寄存器 Sd 的哪一半用于操作数或目标：如果 y 是 B，则 Sm 或 Sd 的低半部分（第15位到第0位）被使用。如果 y 是 T，则 Sm 或 Sd 的高半部分（第31位到第16位）被使用。

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标寄存器
- *'Sm'*是操作数寄存器。

运营

这条带有.F16.32后缀的指令:
1. 将单精度寄存器的顶部或底部半部分中的半精度值转换为单精度值。2. 将结果写入单精度寄存器。

这条指令带有.F32.F16后缀:
1. 将单精度寄存器中的值转换为半精度。2. 将结果写入单精度寄存器的上半部分或下半部分，保留目标寄存器的另一半。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.7 向量除法

对浮点数进行除法运算。

语法

VDIV{cond}.F32 {Sd,} Sn, Sm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标寄存器
- *'Sn, Sm'*是操作数寄存器。

运营

此指令:

1. 将一个浮点数除以另一个浮点数。 2. 将结果写入浮点数目标寄存器。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.8 向量浮点乘加，向量浮点乘减

浮点融合乘加和减法

语法

向量浮点乘法累加{cond}.32位浮点数
{Sd,} Sn, Sm 向量浮点乘法减法{cond
}.32位浮点数 {Sd,} Sn, Sm
其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标寄存器
- *'Sn, Sm'*是操作数寄存器。

运营

VFMA指令：
1. 将操作数寄存器中的浮点值相乘。2. 将结果累加到目标寄存器中。3.
乘法的结果在累加之前不会进行舍入。

VFMS指令：
1. 对第一个操作数寄存器取反。 2. 将第一个和第二个操作数寄存器的浮点值相乘
。 3. 将乘积相加到目标寄存器中。 4. 将结果存入目标寄存器。 5. 乘法结果在相
加之前不进行四舍五入处理。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.9 VFNMA, VFNMS

浮点融合取反乘加和减法

语法

VFNMA{cond}.F32 {Sd,} Sn, Sm VFN
MS{cond}.F32 {Sd,} Sn, Sm

其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标寄存器
- *'Sn, Sm'*是操作数寄存器。

运营

VFNMA指令：
1. 对第一个浮点数操作数寄存器取反。 2. 将第一个浮点数操作数与第二个浮点数操作数相乘。 3. 将浮点数目标寄存器的取反值加到乘积上。 4. 将结果放入目标寄存器。

乘法运算的结果在加法之前不会进行四舍五入。{v*}

VFNMS指令：
1. 将第一个浮点操作数与第二个浮点操作数相乘。2. 将目标寄存器中的浮点值取反后，加到乘积中。3. 将结果放入目标寄存器。

乘法运算的结果在加法之前不会进行四舍五入。{v*}

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.10 VLDM

浮点加载多个 {v*}

语法

VLDM{mode}{cond}{.size} Rn{!}, 列表

其中:

- *'mode'*是寻址方式：IA：增量后。连续的地址从Rn指定的地址开始。DB：减量前。连续的地址在Rn指定的地址之前结束。
- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Size'*是可选的数据大小指定符。
- *'Rn'*是基址寄存器。SP 可以被使用。
- *'!'*这是将修改后的值写回 Rn 的指令的命令。如果模式为 == DB，则这是必需的；如果模式为 == IA，则这是可选的。
- *'list'*是待加载的扩展寄存器列表，作为连续编号的双字或单字寄存器列表，用逗号分隔并用括号包围。

运营

该指令使用来自ARM核心寄存器的地址作为基地址，从连续的内存地址加载多个扩展寄存器。

限制

限制条件如下:

- 如果存在size，则它必须等于列表中寄存器的位数，即32位或64位。
- 对于基地址，可以使用SP。
- 在ARM指令集中，若!未指定，则可使用PC。
- 列表必须包含至少一个寄存器。如果它包含双字寄存器，则不能包含超过16个寄存器。
- 如果使用先递减寻址方式，则写回标志 ! 必须附加在基址寄存器说明中。

条件标志

这些指令不会更改标志。

3.10.11 VLDR

从内存中加载单个扩展寄存器

语法

VLDR{cond}{.64} Dd, [Rn{#imm}] VLDR{cond}{.64} Dd, 标签 VLDR{cond}{.64} Dd, [PC, #imm}] VLDR{cond}{.32} Sd, [Rn {, #imm}] VLDR{cond}{.32} Sd, 标签 VLDR{cond}{.32} Sd, [PC, #imm]

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'64, 32*是可选的数据大小说明符。
- *Dd*是双字加载的目标寄存器。
- *Sd*是单字加载的目标寄存器。
- *Rn*是基址寄存器。堆栈指针可以使用。
- *imm*是 + 或 - 立即偏移量，用于形成地址。 允许的地址值为 0 到 1020 范围内的 4 的倍数。
- *label*是待加载的字面数据项的标签。{v*}

运营

该指令从内存中加载一个扩展寄存器，使用来自ARM核心寄存器的基地址，带有可选的偏移量。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.12 VLMA, VLMS

将两个浮点数相乘，并累加或减去结果。{v*}

语法

向量加载乘法累加{cond}.F32 Sd, S
n, Sm 向量加载乘法求和{cond}.F3
2 Sd, Sn, Sm
其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标浮点值
- *'Sn, Sm'*操作数的浮点值。

运营

浮点乘积累加指令:
1. 乘以两个浮点数。2. 将结果添加到目标浮点数上。

浮点乘减指令:
1. 将两个浮点数相乘。2. 从目标浮点数中减去乘积。

将结果放置在目标寄存器中。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.13 VMOV 立即数

移动浮点立即数

语法

VMOV{cond}.F32 Sd, #imm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是分支目标
- *'imm'*是一个浮点常量。

运营

该指令将一个常量值复制到浮点寄存器。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.14 VMOV 寄存器

将一个寄存器的内容复制到另一个寄存器中。

语法

VMOV{v*}.F64 Dd, Dm VM
OV{v*}.F32 Sd, Sm

其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Dd'*是目标寄存器，用于双字操作。
- *'Dm'*是源寄存器，用于双字操作。
- *'Sd'*是目标寄存器，用于单字操作。{v*}
- *'Sm'*是源寄存器，用于单字操作。

运营

该指令将一个浮点寄存器的内容复制到另一个浮点寄存器。限制

没有限制

条件标志

这些指令不会更改标志。

3.10.15 VMOV 标量到ARM核心寄存器 r

将双字浮点寄存器的一个字传输到Arm核心寄存器。

语法

VMOV{cond} Rt, Dn[x]

其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Rt'*这是目标 Arm 核心寄存器。
- *'Dn'*是64位双字寄存器。
- *'x'*指定使用双字寄存器的哪一半： 如果x为0，使用低半部分；如果x为1，使用高半部分。

运营

该指令将双字浮点寄存器的上半部分或下半部分的一个字传输到Arm核心寄存器。

限制

Rt 不能是 PC 或 SP。

条件标志

这些指令不会更改标志。

3.10.1 6 VMOV ARM核心寄存器到单精度 视图

将单精度寄存器与ARM核心寄存器之间进行传输。

语法

VMOV{cond} Sn, Rt V
MOV{cond} Rt, Sn

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sn'*是单精度浮点数寄存器。
- *'Rt'*是ARM核心寄存器。

运营

此指令转移:

- 单精度寄存器的内容到Arm核心寄存器。
- 将ARM核心寄存器的内容传输到单精度寄存器。

限制

Rt 不能是 PC 或 SP。

条件标志

这些指令不会更改标志。

3.10.17 虚拟机OV 两个ARM核心寄存器到两个单个pr 决定

将两个连续编号的单精度寄存器与两个Arm核心寄存器之间进行传输。

语法

向量移动{cond} Sm, Sm1, Rt, Rt2 向
量移动{cond} Rt, Rt2, Sm, Sm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sm'* 是第一个单精度寄存器。
- *'Sm1'*是一个第二个单精度寄存器（Sm之后的下一个单精度寄存器）
- *'Rt'* 是ARM核心寄存器，Sm是传输到或从其进行传输的。
- *'Rt2'*是ARM核心寄存器，Sm1传输至或自该寄存器。

运营

此指令转移:

1. 两个连续编号的单精度寄存器的内容转移到两个Arm核心寄存器。2. 两个Arm核心寄存器的内容转移到一对单精度寄存器。

限制

限制条件如下:

- 浮点寄存器必须是连续的，依次排列。
- ARM核心寄存器不必是连续的。{v*}
- Rt 不能是 PC 或 SP。

条件标志

这些指令不会更改标志。

3.10.18 VMOV ARM核心寄存器到scala r

将一个字从ARM核心寄存器传输到浮点寄存器。

语法

VMOV{cond}{.32} Dd[x], Rt

其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *32*是一个可选的数据大小指定符。
- *Dd[x]* 是目标，其中 [x] 定义了双字的哪一半被传输，如下：如果 x 为 0，提取低半部分；如果 x 为 1，提取高半部分。
- *Rt*是源 Arm 核心寄存器。

运营

该指令将一个字从Arm核心寄存器传输到双字浮点寄存器的上半部分或下半部分。

限制

Rt 不能是 PC 或 SP。

条件标志

这些指令不会更改标志。

3.10.19 VMRS

从浮点系统寄存器移动到ARM核心寄存器。

语法

VMRS{cond} Rt，FPSCR VMRS{con
d} APSR_nzcv，FPSCR

其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Rt'*是目标ARM核心寄存器。该寄存器可以是R0-R14。
- *'APSR_nzcv'*将浮点标志传输到APSR标志。

运营

此指令执行以下之一操作：

1. 将FPSCR的值复制到通用寄存器中。2. 将FPSCR标志位的值复制到APSR的N、Z、C和V标志中。

限制

Rt 不能是 PC 或 SP。

条件标志

这些指令可选地更改标志位：N, Z, C, V

3月10日20 VMSR

从ARM核心寄存器转移到浮点系统寄存器。

语法

VMSR{cond} FPSCR, Rt

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Rt'*是将被传输到FPSCR的通用寄存器。

运营

该指令将通用寄存器的值移动到FPSCR。参见*Floating-point status control register (FPSCR) on page 255*以获取更多信息。

限制

限制是 Rt 不能为 PC 或 SP。

条件标志

该指令更新FPSCR。

3.10.21 VMUL

浮点乘法。

语法

向量乘法{cond}.F32 {Sd,} Sn, Sm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标浮点值
- *'Sn, Sm'*操作数的浮点值。

运营

此指令:

1. 将两个浮点数相乘。2. 将结果存入目标寄存器

。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.22 VNEG

浮点数取反

语法

取反{cond}.F32 Sd, Sm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标浮点值
- *'Sm'*指的是操作数的浮点值。

运营

此指令:
1. 取反一个浮点值。 2. 将结果存储在第二个浮点寄存器中。 3. 浮点指令翻转符号位。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.23 VNMLA, VNMLS, VNMUL

浮点乘法带取反操作随后进行加法或减法。{v*}

语法

VNMLA{cond}.F32 Sd, Sn, Sm VNMLS
{cond}.F32 Sd, Sn, Sm VNMUL{cond}.
F32 {Sd,} Sn, Sm

其中：

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标浮点值
- *'Sn, Sm'*操作数的浮点值。

运营

VNMLA 指令：
1. 将两个浮点寄存器值相乘。2. 将目标寄存器中的浮点值取反后，将其加到乘积取反的结果上。3. 将结果写回目标寄存器。

VNMLS指令：
1. 浮点寄存器中的两个值相乘。2. 将目标寄存器中的浮点值取反后加到乘积中。3. 将结果写回目标寄存器。

VNMUL指令：
1. 将两个浮点寄存器值相乘。2. 将结果的取反写入目标寄存器。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.24 VPOP

浮点扩展寄存器 弹出.

语法

VPOP{cond}{.size} 列表

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'size'*是一个可选的数据大小指定符。如果存在，它必须等于列表中寄存器的位数，即32位或64位。
- *'list'* 是一个待加载的扩展寄存器列表，作为连续编号的双字或单字寄存器列表，用逗号分隔并用括号包围。

运营

该指令从堆栈中加载多个连续的扩展寄存器。

限制

列表必须包含至少一个寄存器，并且不超过十六个寄存器。

条件标志

这些指令不会更改标志。

3.10.25 VPUSH

浮点扩展寄存器压入

语法

VPUSH{cond}{.size} 列表

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'size'*是一个可选的数据大小指定符。如果存在，它必须等于列表中寄存器的位数，即32位或64位。
- *'list'*是一个待存储的扩展寄存器列表，作为连续编号的双字或单字寄存器列表，用逗号分隔并用括号包围。

运营

该指令将多个连续的扩展寄存器存储到堆栈中。

限制

限制是列表必须包含至少一个寄存器，且不超过十六个。

条件标志

这些指令不会更改标志。

3.10.26 VSQRT

浮点数平方根

语法

向量平方根{cond}.F32 Sd, Sm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标浮点值
- *'Sm'*指的是操作数的浮点值。

运营

此指令:

1. 计算浮点寄存器中值的平方根。2. 将结果写入另一个浮点寄存器。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.10.27 VSTM

浮点存储多个

语法

VSTM{mode}{cond}{.size} Rn{!}, 列表

其中:

- *'mode'* 是寻址方式：IA 执行后递增。连续地址从 Rn 指定的地址开始。这是默认方式，可以省略。DB 执行前递减。连续地址结束于 Rn 指定的地址之前。

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'size'*是一个可选的数据大小指定符。如果存在，它必须等于列表中寄存器的位数，即32位或64位。
- *'Rn'*是基址寄存器。堆栈指针可以使用。
- *'!'* 是导致指令将修改后的值写回 Rn 的功能。如果模式为 == DB 则必需。

- *'list'* 是一个待存储的扩展寄存器列表，作为连续编号的双字或单字寄存器列表，用逗号分隔并用括号包围。

运营

该指令使用来自Arm核心寄存器的基地址，将多个扩展寄存器存储到连续的内存位置。

限制

限制条件如下:

- 列表必须至少包含一个寄存器。
- 如果它包含双字寄存器，则不能包含超过16个寄存器。
- 使用PC作为Rn已被弃用。

条件标志

这些指令不会更改标志。

3.10.28 VSTR

浮点存储

语法

VSTR{cond}{.32} Sd, [Rn{, #imm}] VSTR{c
ond}{.64} Dd, [Rn{, #imm}]

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'32, 64'*是可选的数据大小说明符。
- *'Sd'* 是单字存储的源寄存器。
- *'Dd'* 是双字存储的源寄存器。
- *'Rn'* 是基址寄存器。SP 可以使用。
- *'imm'*是用于形成地址的+或立即偏移量。值为0-1020范围内的4的倍数。可以省略imm，表示偏移量为+0。

运营

该指令将单个扩展寄存器存储到内存中，使用来自Arm核心寄存器的地址，带有可选偏移量，该偏移量在imm中定义。

限制

关于Rn使用PC的限制已弃用。

条件标志

这些指令不会更改标志。

3.10.29 VSUB

浮点数减法 {v*}

语法

向量减法{cond}.F32 {Sd,} Sn, Sm

其中:

- *'cond'* 是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Sd'*是目标浮点值
- *'Sn, Sm'*操作数的浮点值。

运营

此指令:

1. 从一个浮点数中减去另一个浮点数。2. 将结果存入目标浮点数寄存器。

限制

没有限制。

条件标志

这些指令不会更改标志。

3.11 杂项指令

*Table 36*显示剩余的Cortex-M4指令:

表36. 杂项指令

Mnemonic	Brief description	See
BKPT	Breakpoint	*BKPT on page 181*
CPSID	Change Processor State, Disable Interrupts	*CPS on page 182*
CPSIE	Change Processor State, Enable Interrupts	*CPS on page 182*
DMB	Data Memory Barrier	*DMB on page 183*
DSB	Data Synchronization Barrier	*DSB on page 184*
ISB	Instruction Synchronization Barrier	*ISB on page 185*
MRS	Move from special register to register	*MRS on page 186*
MSR	Move from register to special register	*MSR on page 187*
NOP	No Operation	*NOP on page 188*
SEV	Send Event	*SEV on page 189*
SVC	Supervisor Call	*SVC on page 190*
WFE	Wait For Event	*WFE on page 191*
WFI	Wait For Interrupt	*WFI on page 192*

3.11.1 断点

断点。

语法

断点 #imm

其中:

- *'imm'* 是一个用于计算整数的表达式，范围为0-255（8位值）。'

运营

BKPT指令会使处理器进入调试状态。调试工具可以利用此功能在达到特定地址的指令时分析系统状态。

*imm*被处理器忽略。如果需要的话，调试器可以使用它来存储有关断点的附加信息。

BKPT指令可以放置在IT块内，但会无条件执行，不受IT指令指定条件的影响。

条件标志

该指令不会改变标志。

示例

```
BKPT 0xAB  ; 断点，立即值设置为0xAB（调试器 can ）
; 通过使用程序计数器定位立即数来提取立即数
```

3.11.2 CPS

更改处理器状态。

语法

CPSeffect 中断标志

其中:

- *'effect'*属于以下之一：IE: 清除专用寄存器. ID: 设置专用寄存器。
- *'iflags'*这是一个或多个标志的序列：i：设置或清除 PRIMASK. f：设置或清除 FAULTMASK。

运营

CPS会修改PRIMASK和FAULTMASK这两个特殊寄存器的值。如需了解有关这些寄存器的更多信息，请参阅*Exception mask registers on page 23*。

限制

限制条件如下:

- 仅在特权软件中使用CPS，若在非特权软件中使用则无效 {v*}
- CPS不能作为条件，因此不得在IT块内使用。

条件标志

该指令不会改变状态标志。

示例

CPSID i ；禁用中断和可配置的故障处理程序（设置PRIMASK） CPSID f ；禁用中断和所有故障处理程序（设置FAULTMASK） CPSIE i ；启用中断和可配置的故障处理程序（清除PRIMASK） CPSIE f ；启用中断和故障处理程序 （清除FAULTMASK）

3.11.3 数字媒体广播

数据内存屏障

语法

DMB{cond}

其中：*'cond'* 是一个可选的条件代码，参见 *Conditional execution on page 65*。

运营

DMB充当数据内存屏障。它确保所有在程序顺序中出现在DMB指令之前的显式内存访问，在任何在程序顺序中出现在DMB指令之后的显式内存访问完成之前已经完成。DMB不会影响不访问内存的指令的顺序或执行。

条件标志

该指令不会改变标志。

示例

数据内存屏障

3.11.4 双边带

数据同步障碍

语法

DSB{cond}

其中：*'cond'* 是一个可选的条件代码，参见 *Conditional execution on page 65*。

运营

DSB作为一种特殊的数据显示同步内存屏障。在程序顺序中，位于DSB之后的指令不会执行，直到DSB指令完成。DSB指令完成的条件是其之前的所有显式内存访问都已完成。

条件标志

该指令不会改变标志。

示例

```
DSB ; 数据同步屏障
```

3.11.5 国际标准书号

指令同步屏障

语法

国际标准书号{cond}

其中：*'cond'* 是一个可选的条件代码，参见 *Conditional execution on page 65*。

运营

ISB 作用为指令同步屏障。它会刷新处理器的流水线，使得在 ISB 指令执行完毕后，所有后续的指令都会重新从缓存或内存中获取。

条件标志

该指令不会改变标志。

示例

```
ISB ; 指令同步屏障
```

3.11.6 多元回归分析

将特殊寄存器中的内容转移到通用寄存器。

语法

MRS{条件码} Rd, 特殊寄存器

其中:

- *'cond'*是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Rd'*是目标寄存器。
- *'spec_reg'* 可以是以下任意一种：APSR、IPSR、EPSR、IEPSR、IAPSR、EAPSR、PSR、MSP、PSP、PRIMASK、BASEPRI、BASEPRI_MAX、FAULTMASK 或 CONTROL。

运营

将 MRS 与 MSR 结合使用，作为读取-修改-写入序列的一部分，用于更新 PSR，例如清除 Q 标志。参见 *MSR on page 187*.
在进程切换代码中，被切换出的进程的程序员建模状态必须保存，包括相关的PSR内容。同样，被切换入的进程的状态也必须恢复。这些操作使用保存状态的指令序列中的MRS指令和恢复状态的指令序列中的MSR指令。 当与MRS指令一起使用时，BASEPRI_MAX是BASEPRI的别名。

限制

*Rd*不得是 SP 且不得是 PC。

条件标志

该指令不会改变标志。

示例

```
MRS R0, PRIMASK ; 读取 PRIMASK 的值并写入 R0
```

3.11.7 内存系统寄存器

将通用寄存器的内容移动到指定的专用寄存器。

语法

MSR{cond} 特殊寄存器, Rn

其中:

- *'cond'*是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'Rn'* 是源寄存器
- *'spec_reg'* 可以是以下任意一种：APSR、IPSR、EPSR、IEPSR、IAPSR、EAPSR、PSR、MSP、PSP、PRIMASK、BASEPRI、BASEPRI_MAX、FAULTMASK 或 CONTROL。

运营

MSR中的寄存器访问操作取决于特权级别。非特权软件只能访问APSR，参见 *Table 5: APSR bit definitions on page 21*。特权软件可以访问所有特殊寄存器。

在非特权软件中，对PSR中未分配或执行状态位的写入会被忽略。

当您写入 BASEPRI_MAX 时，指令仅在以下任一条件成立的情况下写入 BASEPRI:

- *Rn*不为零且当前的 BASEPRI 值为 0
- *Rn*不为零且小于当前 BASEPRI 值。

参见 *MRS on page 186*。

限制

*Rn*不得是 SP 且不得是 PC。

条件标志

这 i 指令根据值显式地更新标志 e 在 *Rn* 中。

示例

MSR 控制, R1 ; 读取 R1 的值并写入控制寄存器

3.11.8 无操作

无操作。

语法

无操作{cond}

其中：

- *'cond'*是一个可选的条件码，参见 *Conditional execution on page 65*。

运营

NOP不做任何操作。NOP并非一定需要耗时。处理器可能在它到达执行阶段之前将其从流水线中移除。

使用NOP进行填充，例如将下一条指令对齐到64位边界。

条件标志

该指令不会改变标志。

示例

```
NOP ；无操作
```

3.11.9 严重性

发送事件

语法

严重程度{条件}

其中:

- *'cond'*是一个可选的条件码，参见 *Conditional execution on page 65*。

运营

SEV 是一种提示指令，它会向多处理器系统中的所有处理器触发事件信号。它还会将本地事件寄存器设置为 1，参见 *Power management on page 47*。

条件标志

该指令不会改变标志。

示例

```
SEV ; 发送事件
```

3.11.10 SVC

监督调用 {v*}

语法

支持向量分类{cond} #imm

其中:

- *'cond'*是一个可选的条件码，参见 *Conditional execution on page 65*。
- *'imm'* 是一个求值为整数的表达式，范围在0-255之间（8位值）。

运营

SVC指令引发SVC异常。*imm* 会被处理器忽略。如果需要，异常处理程序可以检索它以确定请求的是哪种服务。

条件标志

该指令不会改变标志。

示例

SVC 0x32 ; 监督调用（SVC处理程序可以通过堆栈PC定位来提取立即数 ）

3.11.11 工作流引擎

等待事件。WFE 是一个提示指令。

语法

WFE{cond} 其中：*'cond'* 是一个可选的条件代码，请参见 *Conditional execution on page 65*。

运营

如果事件寄存器为0，WFE将挂起执行，直到以下任一事件发生：

- 一个异常，除非被异常屏蔽寄存器或当前优先级屏蔽
- 如果系统控制寄存器中的SEVONPEND被置位，异常将进入挂起状态。
- 一个调试入口请求（如果调试已启用）
- 由外设或另一处理器在多处理器系统中触发的事件，使用SEV指令。

如果事件寄存器为1，WFE将其清除为0并立即返回。

如需更多信息，请参见 *Power management on page 47*。

条件标志

该指令不会改变标志。

示例

```
WFE ; 等待事件
```

3.11.12 世界金融研究所

等待中断

语法

WFI{cond}

其中：

- *'cond'*是一个可选的条件码，参见 *Conditional execution on page 65*。

运营

WFI 是一个提示指令，它会暂停执行，直到以下其中一个事件发生：

- 一个例外
- 一个Debug Entry请求，无论Debug是否启用。

条件标志

该指令不会改变标志。

示例

```
WFI ; 等待中断
```

四 核心外设

4.1 关于 STM32 Cortex-M4 核心外设

的*Private peripheral bus*（PPB）地址映射是:

表 37. STM32 核心外设寄存器区域

Address	Core peripheral	Description
0xE000E010-0xE000E01F	System timer	*Table 55 on page 251*
0xE000E100-0xE000E4EF	Nested vectored interrupt controller	*Table 49 on page 219*
0xE000ED00-0xE000ED3F	System control block	*Table 53 on page 244*
0xE000ED88-0xE000ED8B	Floating point unit coprocessor access control	*Table 56 on page 252*
0xE000ED90-0xE000EDB8	Memory protection unit	*Table 44 on page 206*
0xE000EF00-0xE000EF03	Nested vectored interrupt controller	*Table 49 on page 219*
0xE000EF30-0xE000EF44	Floating point unit	*Table 56 on page 252*

在寄存器描述中,

- 寄存器类型描述如下：– RW：读写。
  – RO：只读。– WO：只写。

- *Required privilege*说明访问寄存器所需的特权级别，如下： – 特权：只有特权软件可以访问寄存器。 – 非特权：非特权软件和特权软件都可以访问寄存器。{v*}

4.2 内存保护单元（MPU）

本节描述了某些STM32微控制器中实现的内存保护单元（MPU）。请参考相应的设备数据手册，以确认您使用的STM32型号是否包含MPU。

MPU将内存映射划分为若干区域，并定义每个区域的位置、大小、访问权限和内存属性。它支持:

- 每个区域的独立属性设置
- 重叠区域
- 将内存属性导出到系统中。

内存属性会影响对该区域的内存访问行为。Cortex-M4 MPU定义:

- 八个独立的内存区域，0-7
- 一个背景区域。

当内存区域重叠时，内存访问会受到编号最高的区域属性的影响。例如，区域7的属性优先于任何与区域7重叠的区域的属性。

背景区域具有与默认内存映射相同的内存访问属性，但仅可由特权软件访问。

Cortex-M4 内存保护单元的内存映射是统一的。这意味着指令访问和数据访问具有相同的区域设置。

如果程序访问被MPU禁止的内存位置，处理器将生成内存管理故障。这将引发故障异常，并可能导致操作系统环境中的进程终止。

在操作系统环境中，内核可以根据待执行的进程动态更新MPU区域设置。通常，嵌入式操作系统使用MPU进行内存保护。

MPU区域的配置基于内存类型，参见 *Section 2.2.1: Memory regions, types and attributes on page 29*。

*Table 38*显示了可能的MPU区域属性。

表 38. 内存属性摘要

Memory type	Shareability	Other attributes	Description
Strongly-ordered	-	-	All accesses to Strongly-ordered memory occur in program order. All Strongly-ordered regions are assumed to be shared.
Device	Shared	-	Memory-mapped peripherals that several processors share.
	Non-shared	-	Memory-mapped peripherals that only a single processor uses.
Normal	Shared	Non-cacheable Write-through Cacheable Write-back Cacheable	Normal memory that is shared between several processors.
	Non-shared	Non-cacheable Write-through Cacheable Write-back Cacheable	Normal memory that only a single processor uses.

4.2.1 内存保护单元访问权限属性

本节描述MPU访问权限属性。MPU_RASR寄存器的访问权限位TEX、C、B、S、AP和XN控制对相应内存区域的访问。如果对没有所需权限的内存区域进行访问，则MPU会生成权限故障。

*Table 39*显示了TEX、C、B和S访问权限位的编码。

表 39. TEX、C、B 和 S 编码

TEX	C	B	S	Memory type	Shareability	Other attributes
b000	0	0	x (1)	Strongly-ordered	Shareable	-
		1	x(1)	Device	Shareable	-
	1	0	0	Normal	Not shareable	Outer and inner write-through. No write allocate.
			1		Shareable	
		1	0	Normal	Not shareable	Outer and inner write-back. No write allocate.
			1		Shareable	
b001	0	0	0	Normal	Not shareable	Outer and inner noncacheable.
		-	1		Shareable	
		1	x(1)	Reserved encoding		-
	1	0	x(1)	Implementation defined attributes.		-
		1	0	Normal	Not shareable	Outer and inner write-back. Write and read allocate.
			1		Shareable	
b010	0	0	x(1)	Device	Not shareable	Nonshared Device.
		1	x(1)	Reserved encoding		-
	1	x(1)	x(1)	Reserved encoding		-
b1BB	A	A	0	Normal	Not shareable	Cached memory(2) , BB = outer policy, AA = inner policy.
			1		Shareable	

1. MPU忽略该位的值。
2. 关于AA和BB位的编码，请参见*Table 40*。

*Table 40*显示内存属性编码方式的缓存策略，其中TEX值位于4-7范围内。

表40. 内存属性编码的缓存策略

Encoding, AA or BB	Corresponding cache policy
00	Non-cacheable
01	Write back, write and read allocate
10	Write through, no write allocate
11	Write back, no write allocate

*Table 41*显示了定义特权软件和非特权软件访问权限的AP编码。

表格 41. AP 编码

AP[2:0]	Privileged permissions	Unprivileged permissions	Description
000	No access	No access	All accesses generate a permission fault
001	RW	No access	Access from privileged software only
010	RW	RO	Writes by unprivileged software generate a permission fault
011	RW	RW	Full access
100	Unpredictable	Unpredictable	Reserved
101	RO	No access	Reads by privileged software only
110	RO	RO	Read only, by privileged or unprivileged software
111	RO	RO	Read only, by privileged or unprivileged software

4.2.2 内存保护单元不匹配

当访问违反MPU权限时，处理器会生成内存管理故障，请参见 *Section 2.1.4: Exceptions and interrupts on page 26*。MMFSR指示故障的原因。如需更多信息，请参见*Section 4.4.15: Memory management fault address register (MMFAR) on page 242*。

4.2.3 更新内存保护单元区域

要更新MPU区域的属性，请更新MPU_RNR、MPU_RBAR和MPU_RASR寄存器。您可以分别编程每个寄存器，或使用多字写入操作同时编程所有这些寄存器。您可以使用MPU_RBAR和MPU_RASR的别名，通过STM指令同时编程最多四个区域。

使用逐字更新MPU区域

简单代码配置一个区域:

```
                                    ; R1 = 区域编号 ; R2 = 大小
                                    /启用 ; R3 = 属性 ; R4 = 地
                                    址

LDR R0, =内存保护单元区域号寄存器 ; 0xE000ED98, MPU区域号寄存器
存储R1到[R0, #0x0]          区域编号
将 R4 存储到 [R0, #0x4]     ; 区域基地址
存储半字 R2, [R0, #0x8]     区域大小和启用
存储半字 R3 到 [R0, #0xA]   区域属性
```

如果您之前已启用正在更改的区域，请在向MPU写入新的区域设置之前禁用该区域。例如:

```
                                    ; R1 = region number
                                    ; R2 = size/enable
```

```
                                        ; R3 = 属性 ; R4 = 地址

LDR R0,=MPU_RNR          ; 0xE000ED98, MPU区域号寄存器
存储R1到[R0, #0x0]       区域编号
BIC R2, R2, #1           ;  禁用
存储半字 R2, [R0, #0x8]  ; 区域大小和启用
将 R4 存储到 [R0, #0x4]  ; 区域基地址
存储半字 R3 到 [R0, #0xA] 区域属性
将 R2 与 #1 按位或，结果存入R2 ;启用
存储半字 R2, [R0, #0x8]  ; 区域大小和启用
```

软件必须使用内存屏障指令:

- 在进行MPU配置之前，如果存在未完成的内存传输，如缓冲写入，这些传输可能会受到MPU设置更改的影响
- 在完成MPU配置后，如果该配置包含必须使用新的MPU设置的内存传输。

然而，如果MPU配置过程通过进入异常处理程序开始，或随后发生异常返回，则不需要内存屏障指令，因为异常入口和异常返回机制会引发内存屏障行为。

在MPU设置过程中，软件不需要任何内存屏障指令，因为它通过PPB访问MPU，而PPB是一个强序内存区域。

例如，如果您希望所有内存访问行为在编程序列之后立即生效，请使用DSB指令和ISB指令:

- 在更改MPU设置后需要执行DSB，例如在上下文切换结束时。
- 如果编程MPU区域或区域的代码通过分支或调用进入，则需要使用ISB。如果编程序列通过从异常返回或引发异常进入，则不需要使用ISB。

使用多字写入更新MPU区域

您可以直接使用多字写入进行编程，根据信息被分割的方式。考虑以下重新编程:

```
                                        ; R1 = 区域编号 ; R2 = 地址 ; R3 = 大小，属性在一条L
DR指令中 R0, =MPU_RNR ; 0xE000ED98, MPU区域编号寄存器 STR R1, [R0, #0x0] ;
区域编号 STR R2, [R0, #0x4] ; 区域基地址 STR R3, [R0, #0x8] ; 区域属性，大小和使
能
```

使用STM指令来优化这个:

```
                                        ; R1 = 区域编号 ; R2 = 地址 ; R3 = 大小,
                                        属性在一个

ldr R0, =M     PU_RNR ; 0xE000ED98, MPU区域号寄存器
存储 R0, {R    1-R3} ; 区域编号、地址、属性、大小和使能
```

您可以在两个字中处理预打包的信息。这意味着RBAR包含所需的区域号，并且有效位被设置为1，请参见 *MPU region base address register (MPU_RBAR) on page 203*。当数据是静态打包时使用此方法，例如在引导加载程序中:

```
                                        ; R1 = 地址和区域编号合并为一个字段 ; R2 = 大小和
                                        属性合并为一个字段
LDR R0, =内存保护单元基址寄存器          ; 内存保护单元区域基地址寄存器0xE000ED9C
STR R1, [R0, #0x0]                      ； 区域基地址和
                                        ; 区域号与有效位（第4位）组合，置为1
存储寄存器 R2, [R0, #0x4]               区域属性、大小和启用
```

使用STM指令来优化这个：

```
                                        ; R1 = 地址和区号合并为一个 ; R2 = 大小和属性合并
                                        为一个
加载 R0,=内存保护单元基址寄存器          ; 内存保护单元区域基地址寄存器0xE000ED9C
存储 R0, {R1-R2}                        ； 区域基地址、区域号和VALID位，； 以及区域属性、大小和使能
```

子区域

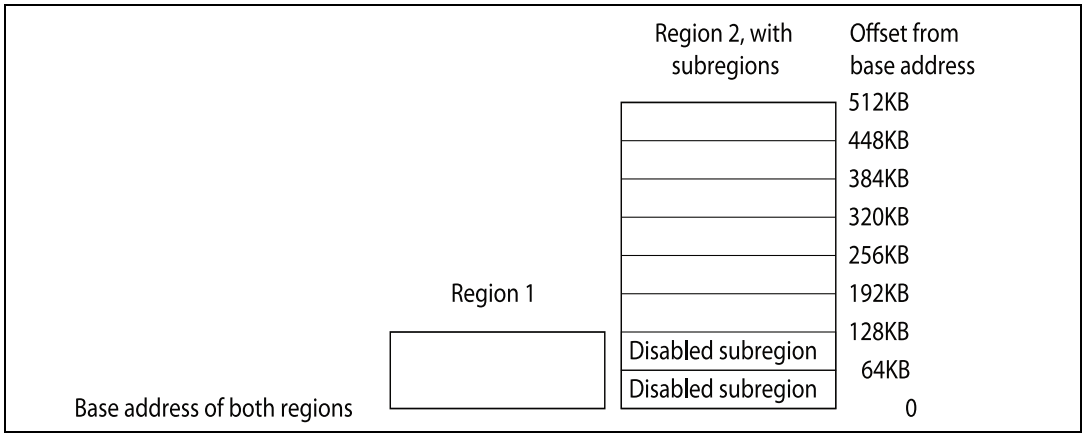
256字节或更大的区域被划分为八个等大小的子区域。在RASR的SRD字段中设置相应的位以禁用子区域，参见*Section 4.2.9: MPU region attribute and size register (MPU_RASR) on page 204*。SRD的最低有效位控制第一个子区域，而最高有效位控制最后一个子区域。禁用子区域意味着另一个与禁用范围重叠的区域匹配。如果没有任何其他启用的区域与禁用的子区域重叠，MPU将引发故障。

32字节、64字节和128字节的区域不支持子区域，对于这些大小的区域，您必须将SRD字段设置为0x00，否则MPU的行为是不可预测的。

SRD使用示例：

两个具有相同基地址的区域重叠。区域一为128KB，区域二为512KB。为确保区域一的属性适用于第一个128KB区域，需将区域二的SRD字段设置为b00000011以禁用前两个子区域，如图所示。

图18. 子区域示例



4.2.4 内存保护单元设计提示与技巧

为了避免意外行为，请在更新可能被中断处理程序访问的区域属性之前禁用中断。

确保软件使用正确大小的对齐访问以访问MPU寄存器：

- 除了RASR之外，它必须使用对齐的字访问。
- 对于RASR，它可以使用字节、对齐的半字或字的访问方式。

处理器不支持对MPU寄存器的非对齐访问。

在设置MPU时，如果MPU之前已被编程，请禁用未使用的区域，以防止任何之前的区域设置影响新的MPU配置。

推荐的MPU配置

STM32微控制器系统只有一个处理器，因此应按照以下方式编程MPU：

表42. STM32内存区域属性

Memory region	TEX	C	B	S	Memory type and attributes
Flash memory	b000	1	0	0	Normal memory, Non-shareable, write-through
Internal SRAM	b000	1	0	1	Normal memory, Shareable, write-through
External SRAM	b000	1	1	1	Normal memory, Shareable, write-back, write-allocate
Peripherals	b000	0	1	1	Device memory, Shareable

在STM32实现中，可共享性及缓存策略属性不会影响系统行为。然而，将这些设置应用于MPU区域可以使应用程序代码更具可移植性。给出的值适用于典型情况。

Note: The MPU attributes don't affect DMA data accesses to the memory/peripherals address spaces. therefore, in order to protect the memory areas against inadvertent DMA accesses, the MPU must control the SW/CPU access to the DMA registers.

4.2.5 MPU类型寄存器（MPU_TYPER）

地址偏移量: 0x00

复位值: 0x0000 0800

所需权限：特权

MPU_TYPER寄存器指示内存保护单元是否存在，如果存在，则它支持多少个内存区域。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								IREGION[7:0]							
								r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DREGION[7:0]							Reserved								SEPARATE
r	r	r	r	r	r	r									r

位31:24保留。

位23:16 IREGION[7:0]：MPU指令区域数量。这些位表示支持的MPU指令区域数量。始终为0x00。MPU内存映射是统一的，由DREGION字段描述。

位15:8 DREGION[7:0]：内存保护单元数据区域数量。 这些位表示支持的内存保护单元数据区域数量。 0x08：八个内存保护单元区域 0x00：内存保护单元不存在

位7:1保留。

位0 SEPARATE：分离标志。此位指示对统一或分离的指令和数据存储器映射的支持：0 = 统一 1 = 分离

4.2.6 MPU控制寄存器 (MPU_CTRL)

地址偏移量：0x04

复位值：0x0000 0000

所需权限：特权

MPU_CTRL寄存器：

- 启用内存保护单元
- 启用默认的内存映射背景区域
- 启用MPU，当处于硬故障、不可屏蔽中断（NMI）和FAULTMASK升级的处理程序中。

当 ENABLE 和 PRIVDEFENA 都被设置为 1 时：

- 对于特权访问，默认内存映射如*Section 2.2: Memory model on page 28*中所述。任何未针对启用内存区域的特权软件访问均按照默认内存映射的定义执行。
- 任何非特权软件对未启用的内存区域的访问会导致内存管理故障。

XN和强有序规则始终适用于系统控制空间，无论启用位的值如何。

当ENABLE位被设置为1时，除非PRIVDEFENA位也被设置为1，否则系统要正常运行必须启用内存映射中的至少一个区域。如果PRIVDEFENA位被设置为1且没有启用任何区域，则只有特权软件才能运行。

当ENABLE位设置为0时，系统使用默认内存映射。这与未实现MPU时的内存属性相同，见*Table 13: Memory access behavior on page 30*。默认内存映射适用于特权软件和非特权软件的访问。

当MPU被启用时，对系统控制空间和向量表的访问始终被允许。其他区域的访问则根据区域设置以及PRIVDEFENA是否设置为1来决定。{v*}

除非HFNMIENA设置为1，当处理器执行优先级为–1或–2的异常处理程序时，MPU未启用。这些优先级仅在处理硬故障或NMI异常，或FAULTMASK被启用时才可能出现。将HFNMIENA位设置为1可启用MPU，当使用这两个优先级时。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													PRIVDEFENA	HFNMIENA	ENABLE
													rw	rw	rw

位31:3 保留，被硬件强制设为0。

位2 PRIVDEFENA：启用特权软件对默认内存映射的访问。 0：如果MPU被启用，则禁用默认内存映射的使用。任何未被任何启用区域覆盖的内存访问将导致故障。 1：如果MPU被启用，则启用默认内存映射作为特权软件访问的背景区域。

Note: When enabled, the background region acts as if it is region number -1. Any region that is defined and enabled has priority over this default map.
If the MPU is disabled, the processor ignores this bit.

位1 HFNMIENA：在硬故障、非屏蔽中断和故障屏蔽处理程序期间使能MPU的操作。

当MPU被启用时：0：在硬故障、NMI和FAULTMASK处理程序中，MPU被禁用，无论ENABLE位的值如何。1：在硬故障、NMI和FAULTMASK处理程序中，MPU被启用。

Note: When the MPU is disabled, if this bit is set to 1 the behavior is unpredictable.

位 0 ENABLE：启用 MPU 0：MPU 禁用 1：MPU 启用

4.2.7 MPU区域号寄存器 (MPU_RNR)

地址偏移量：0x08

复位值：0x0000 0000

所需权限：特权

MPU_RNR寄存器确定由MPU_RBAR和MPU_RASR寄存器引用的内存区域。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								REGION[7:0]							
								rw	rw	rw	rw	rw	rw	rw	rw

位31:8 保留，被硬件强制设为0。

位7:0 REGION[7:0]: MPU区域 这些位指示由MPU_RBAR和MPU_RASR寄存器所引用的MPU区域。 MPU支持8个内存区域，因此该字段的允许值为0-7。通常，在访问MPU_RBAR或MPU_RASR之前，您需要将所需的区域号写入此寄存器。然而，您可以通过将VALID位设置为1后向MPU_RBAR寄存器写入区域号来更改区域号，参见*MPU region base address register (MPU_RBAR)*。此写入操作会更新REGION字段的值。

4.2.8 MPU区域基地址寄存器 (MPU_R 巴

地址偏移量：0x0C

复位值：0x0000 0000

所需权限：特权

MPU_RBAR寄存器定义由MPU_RNR寄存器选择的MPU区域的基地址，并可以更新MPU_RNR寄存器的值。

向MPU_RBAR寄存器写入数据，并将VALID位设置为1，以更改当前区域号并更新MPU_RNR寄存器。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADDR[31:N]...															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
....ADDR[31:N]											VALID	REGION[3:0]			
											rw	rw	rw	rw	rw

位31:N ADDR[31:N]: 区域基地址字段

N的值取决于区域大小。

区域大小由MPU_RASR中的SIZE字段指定，定义了N的值： N = Log2（区域大小（字节））

当区域大小配置为4 GB时，在MPU_RASR寄存器中没有有效的ADDR字段。在这种情况下，该区域占据整个内存映射，基地址为0x00000000。

基地址对齐到区域的大小。例如，一个64 KB的区域必须对齐在64 KB的倍数上，例如在0x00010000或0x00020000处。

位 N-1:5 保留，由硬件强制设为 0。

位4有效：MPU区域编号有效

写：

0: MPU_RNR寄存器未更改，处理器：– 更新由MPU_RNR指定区域的基地址 – 忽略REGION字段的值

1: 处理器： 更新MPU_RNR的值为REGION字段的值 更新REGION字段指定的区域的基地址

阅读：

始终读作零。{v*}

位3:0 REGION[3:0]: MPU区域字段

关于写入行为，请参阅 VALID 字段的描述。在读取时，返回当前区域号，如 MPU_RNR 寄存器所指定。

4.2.9 兆帕 U区域属性和大小寄存器 (MPU_R 自动语音识别)

地址偏移量: 0x10

复位值：0x0000 0000

所需权限：特权

MPU_RASR寄存器定义由MPU_RNR指定的MPU区域的区域大小和内存属性，并启用该区域及其任何子区域。

MPU_RASR 可通过字或半字访问方式进行访问:

- 最显著的半字包含区域属性
- 最低有效半字包含区域大小以及区域和子区域的使能位。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved			XN	deyreseR	AP[2:0]			Reserved		TEX[2:0]			S	C	B
			rw		rw	rw	rw			rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SRD[7:0]								Reserved		SIZE					ELBANE
rw	rw	rw	rw	rw	rw	rw	rw			rw	rw	rw	rw	rw	rw

位31:29 保留，被硬件强制设为0。

位28 XN：指令访问禁用位：0：启用指令获取 1：禁用指令获取。位27 保留，硬件强制为0。位26:24 AP[2:0]：访问权限 关于访问权限的信息，请参阅 *Section 4: Core peripherals*。AP位编码的描述，请参阅 *Table 41 on page 196* 。

位23:22 保留，由硬件强制设置为0。

位21:19 TEX[2:0]：内存属性 关于TEX位编码的描述，请参见 *Table 39 on page 195* 位18 S：可共享的内存属性 关于S位编码的描述，请参见 *Table 39 on page 195* 位17 C：内存属性 位16 B：内存属性

15:8位 SRD：子区域禁用位。对于该字段中的每一位：0：对应的子区域被启用；1：对应的子区域被禁用。详见 *Subregions on page 198*。128字节及以下的区域不支持子区域。当为此类区域编写属性时，将SRD字段写为0x00。

位7:6 保留，被硬件强制置为0。

位5:1 SIZE: MPU保护区域的大小。最小允许值为3（b00010），详见*SIZE field values*。

位0 使能位：区域使能位。

大小字段值

SIZE字段定义了由MPU_RNR寄存器指定的MPU内存区域的大小，如下所示：

(区域大小（字节）) = 2(SIZE+1)

允许的最小区域大小为32B，对应于SIZE值为4。*Table 43* 提供了示例SIZE值，以及MPU_RBAR中对应的区域大小和N的值。

表43。示例 SIZE 字段值

SIZE value	Region size	Value of N(1)	Note
b00100 (4)	32B	5	Minimum permitted size
b01001 (9)	1KB	10	-
b10011 (19)	1MB	20	-
b11101 (29)	1GB	30	-
b11111 (31)	4GB	b01100	Maximum possible size

1. 在MPU_RBAR寄存器中查看 *Section 4.2.8 on page 203*

4.2.10 内存保护单元寄存器映射

表 44. MPU寄存器映射和复位值

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	MPU_TYPER	Reserved								IREGION[7:0]								DREGION[7:0]								Reserved									ETARAPES
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0x04	MPU_CTRL	Reserved																													ANEFEDV_RP	ANE_MNFH	ELBANE
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	MPU_RNR	Reserved																								REGION[7:0]							
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	MPU_RBAR	ADDR[31:N]...																											DLAV	NO_GER[3:0]			
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	MPU_RASR	reserved			XN	reserved	AP[2:0]			reserved	TEX[2:0]			S	C	B	SRD[7:0]								reserved	SIZE						ELBANE
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x14	MPU_RBAR_A1 (1)	ADDR[31:N]...																											DLAV	NO_GER[3:0]			
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	MPU_RASR_A1 (2)	reserved			XN	reserved	AP[2:0]			reserved	TEX[2:0]			S	C	B	SRD[7:0]								reserved	SIZE						ELBANE
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1C	MPU_RBAR_A2 (1)	ADDR[31:N]...																											DLAV	NO_GER[3:0]			
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x20	MPU_RASR_A2 (2)	reserved			XN	reserved	AP[2:0]			reserved	TEX[2:0]			S	C	B	SRD[7:0]								reserved	SIZE						ELBANE
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

表44. MPU寄存器映射及复位值（续 尤德)

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1C	MPU_RBAR_A3 (1)	ADDR[31:N]...																											DELAV	REGION[3:0]			
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x20	MPU_RASR_A3 (2)	reserved				XN	reserved	AP[2:0]			reserved	TEX[2:0]			S	C	B	SRD[7:0]								reserved	SIZE					ENABLE
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1. MPU_RBAR寄存器的别名 2.
MPU_RASR寄存器的别名

4.3 N 测试的向量中断控制器 (N 维克)

本节描述嵌套向量中断控制器（NVIC）以及其使用的寄存器。NVIC 支持：

- 最多240个中断
- 每个中断的可编程优先级等级为{v*}。较高的等级对应较低的优先级，因此等级0是最高中断优先级。
- 中断信号的电平和脉冲检测
- 中断的动态重新调整优先级
- 将优先级值分组为组优先级和子优先级字段
- 中断尾链
- 一个外部的 *Non-maskable interrupt* (NMI)

处理器在异常入口时自动将其状态压栈，并在异常出口时弹出该状态，无需指令开销。这提供了低延迟的异常处理。 NVIC寄存器的硬件实现是：

表45. NVIC寄存器摘要

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100-0xE000E11F	NVIC_ISER0-NVIC_ISER7	RW	Privileged	0x00000000	*Table 4.3.2: Interrupt set-enable register x (NVIC_ISERx) on page 210*
0XE000E180-0xE000E19F	NVIC_ICER0-NVIC_ICER7	RW	Privileged	0x00000000	*Table 4.3.3: Interrupt clear-enable register x (NVIC_ICERx) on page 211*
0XE000E200-0xE000E21F	NVIC_ISPR0-NVIC_ISPR7	RW	Privileged	0x00000000	*Table 4.3.4: Interrupt set-pending register x (NVIC_ISPRx) on page 212*
0XE000E280-0xE000E29F	NVIC_ICPR0-NVIC_ICPR7	RW	Privileged	0x00000000	*Table 4.3.5: Interrupt clear-pending register x (NVIC_ICPRx) on page 213*
0xE000E300-0xE000E31F	NVIC_IABR0-NVIC_IABR7	RW	Privileged	0x00000000	*Table 4.3.6: Interrupt active bit register x (NVIC_IABRx) on page 214*
0xE000E400-0xE000E4EF	NVIC_IPR0-NVIC_IPR59	RW	Privileged	0x00000000	*Table 4.3.7: Interrupt priority register x (NVIC_IPRx) on page 215*
0xE000EF00	STIR	WO	Configurable	0x00000000	*Table 4.3.8: Software trigger interrupt register (NVIC_STIR) on page 216*

Note: The number of interrupts is product-dependent. Refer to reference manual/datasheet of relevant STM32 product for related information.

4.3.1 使用CMSIS访问Cortex-M4 NVIC寄存器

CMSIS函数使软件能够在不同的Cortex-M配置处理器之间实现可移植性。在使用CMSIS时，要访问NVIC寄存器，请使用以下函数：

表46. 通过CMSIS访问NVIC功能

CMSIS function(1)	Description
void NVIC_EnableIRQ(IRQn_Type IRQn)	Enables an interrupt or exception.
void NVIC_DisableIRQ(IRQn_Type IRQn)	Disables an interrupt or exception.
void NVIC_SetPendingIRQ(IRQn_Type IRQn)	Sets the pending status of interrupt or exception to 1.
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)	Clears the pending status of interrupt or exception to 0.
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)	Sets the priority of an interrupt or exception with configurable priority level to 1.
uint32_t NVIC_GetPriority(IRQn_Type IRQn)	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.

1. 输入参数 IRQn 是中断号。可能的'n'值取决于产品。请参阅相关STM32产品的参考手册/数据手册以获取相关信息。

4.3.2 中断使能寄存器 x (NVIC_ISERx)

地址偏移量：0x100 + 0x04 * x, (x = 0 到 7) 复位值：0x0000 0000 所需权限：特权级 NVIC_ISER0 的第 0 至 31 位分别对应中断 0 至 31，分别 NVIC_ISER1 的第 0 至 31 位分别对应中断 32 至 63，分别 …… NVIC_ISER6 的第 0 至 31 位分别对应中断 192 至 223，分别 NVIC_ISER7 的第 0 至 15 位分别对应中断 224 至 239，分别

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETENA[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETENA[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

位31:0 SETENA: 中断设置使能位。写入：0: 无影响 1: 使能中断

阅读:

0: 中断禁用 1: 中断启用.

如果一个待处理中断已启用，NVIC 将根据其优先级激活该中断。如果一个中断未启用，断言其中断信号会将中断状态更改为待处理，但无论其优先级如何，NVIC 都不会激活该中断。NVIC_ISER7 寄存器的第16到31位保留。

Note: The number of interrupts is product-dependent. Refer to reference manual/datasheet of relevant STM32 product for related information.

4.3.3 中断清除使能寄存器 x (NVIC_ICER x）

地址偏移量：0x180 + 0x04 * x，(x = 0 到 7) 复位值：0x0000 0000
所需权限：特权 NVIC_ICER0 位 0 到 31 分别对应中断 0 到 31，特权 NVIC_ICER1 位 0 到 31 分别对应中断 32 到 63，…… 特权 NVIC_ICER6 位 0 到 31 分别对应中断 192 到 223，特权 NVIC_ICER7 位 0 到 15 分别对应中断 224 到 239

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CLRENA[31:16]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLRENA[15:0]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

位31:0 CLRENA：中断清除使能位。 写入：0：无影响 1：禁用中断 读取：0：中断禁用 1：中断使能。 NVIC_ICER7寄存器的位16到31保留。

Note: The number of interrupts is product-dependent. Refer to reference manual/datasheet of relevant STM32 product for related information.

4.3.4 中断设置为挂起寄存器x (NVIC_ISPR) x）

地址偏移量：0x200 + 0x04 * x，(x = 0 到 7) 复位值：0x0000 0000
所需权限：特权 NVIC_ISPR0 的位 0 到 31 分别对应中断 0 到 31，NVIC_ISPR1 的位 0 到 31 分别对应中断 32 到 63，... NVIC_ISPR6 的位 0 到 31 分别对应中断 192 到 223，NVIC_ISPR7 的位 0 到 15 分别对应中断 224 到 239

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETPEND[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETPEND[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

位 31:0 SETPEND: 中断设置挂起位

写入: 0: 无效果 1: 修改中断状态为待处理

阅读:

0: 中断未挂起 1: 中断已挂起 向对应于已挂起中断的ISPR位写入1没有效果。向对应于禁用中断的ISPR位写入1会将该中断的状态设置为挂起。NVIC_ISPR7寄存器的位16到31保留。

Note: The number of interrupts is product-dependent. Refer to reference manual/datasheet of relevant STM32 product for related information.

4.3.5 中断清除挂起寄存器 x (NVIC_IC PRx)

地址偏移量: 0x280 + 0x04 * x, (x = 0到7) 复位值: 0x0000 0000 所需权限: 特权 NVIC_ICPR0 的位 0 到 31 分别对应中断 0 到 31，分别 NVIC_ICPR1 的位 0 到 31 分别对应中断 32 到 63，分别 .... NVIC_ICPR6 的位 0 到 31 分别对应中断 192 到 223，分别 NVIC_ICPR7 的位 0 到 15 分别对应中断 224 到 239，分别

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CLRPEND[31:16]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CLRPEND[15:0]															
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

位 31:0 CLRPEND：中断清除挂起位

写入：0：无效果 1：清除中断的挂起状态

阅读:

0：中断未挂起 1：中断已挂起 将1写入ICPR位不会影响相应中断的活动状态。NVIC_ICPR7寄存器的位16到31保留。

Note: The number of interrupts is product-dependent. Refer to reference manual/datasheet of relevant STM32 product for related information.

4.3.6 中断活动位寄存器x (NVIC_IABRx)

地址偏移量：0x300 + 0x04 * x，(x = 0到7) 复位值：0x0000 0000 所需权限：特权 NVIC_IABR0 的位 0 到 31 分别对应中断 0 到 31，分别 NVIC_IABR1 的位 0 到 31 分别对应中断 32 到 63，分别 …… NVIC_IABR6 的位 0 到 31 分别对应中断 192 到 223，分别 NVIC_IABR7 的位 0 到 15 分别对应中断 224 到 239，分别

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ACTIVE[31:16]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ACTIVE[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:0 活动: 中断活动标志

0: 中断未激活 1: 中断激活 一位读为1，如果对应中断的状态为激活或激活且待处理。NVIC_IABR7寄存器的第16到31位保留。

Note: The number of interrupts is product-dependent. Refer to reference manual/datasheet of relevant STM32 product for related information.

4.3.7 中断优先级寄存器 x (NVIC_IPRx)

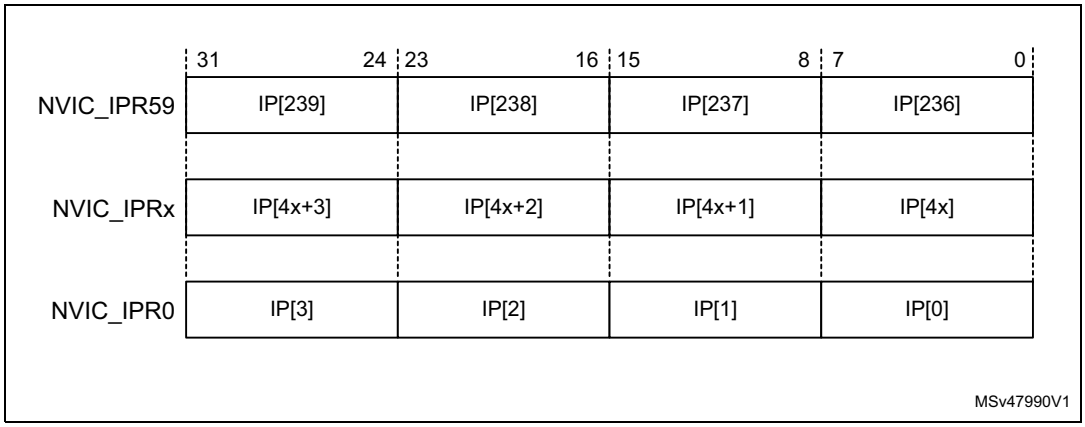
地址偏移量: 0x400 + 0x04 × x, (x = 0到59)

复位值：0x0000 0000

所需权限：特权

NVIC_IPRx（x = 0到59）字节可访问的寄存器提供8位优先级字段IP[N]（N = 0到239），用于每个中断的240个中断。每个寄存器包含四个IP[N]字段，如*Figure 19*所示。

图19. NVIC_IPRx寄存器中IP[N]字段的映射



下表显示任何NVIC_IPRx寄存器的位分配。每个IP[N]字段的顺序可以表示为N = 4 * x +字节偏移量。

表47. NVIC_IPRx 位分配

Bits	Name	Function
[31:24]	Priority, byte offset = 3	Each priority field holds a priority value, 0-255. The lower the value, the greater the priority of the corresponding interrupt. The processor implements only bits[7:4] of each field, bits[3:0] read as zero and ignore writes.
[23:16]	Priority, byte offset = 2	
[15:8]	Priority, byte offset = 1	
[7:0]	Priority, byte offset = 0	

参见 *Interrupt set-enable register x (NVIC_ISERx) on page 210* 以了解从软件角度的中断优先级。

Note: The number of interrupts is product-dependent. Refer to reference manual/datasheet of relevant STM32 product for related information.

4.3.8 软件触发中断寄存器 (NVIC_ST IR)

地址偏移量：0xE00

复位值：0x0000 0000

所需特权：当SCR中的USERSETMPEND位被设置为1时，非特权软件可以访问STIR，参见*Section 4.4.6: System control register (SCR)*。只有特权软件可以启用对STIR的非特权访问。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							INTID[8:0]								
							w	w	w	w	w	w	w	w	w

位31:9 保留，必须保持清除。

位 8:0 INTID 软件生成的中断ID 写入 STIR 以生成软件生成中断（SGI）。需要写入的值是所需 SGI 的中断ID，范围为 0-239。例如，0x03 的值指定中断 IRQ3。

4.3.9 电平敏感和脉冲中断

STM32中断既电平敏感又脉冲敏感。脉冲中断也被称为边沿触发中断。

电平敏感中断保持激活状态，直到外设取消中断信号。通常，这是由于中断服务程序访问外设，导致其清除中断请求。脉冲中断是在处理器时钟的上升沿同步采样的中断信号。为确保NVIC检测到中断，外设必须激活中断信号至少一个时钟周期，在此期间，NVIC检测到脉冲并锁存中断。

当处理器进入中断服务例程时，它会自动将中断的待处理状态清除，参见 *Hardware and software control of interrupts*。对于水平触发中断，如果处理器从中断服务例程返回之前，信号未被释放，则中断会再次变为待处理状态，处理器必须再次执行其中断服务例程。这意味着外设可以保持中断信号有效，直到其不再需要服务。{v*}

硬件和软件的中断控制

Cortex-M4锁存所有中断。一个外设中断会因为以下原因之一而待处理：

- NVIC 检测到中断信号为 HIGH 且中断未激活
- NVIC 检测中断信号的上升沿
- 软件向对应的中断置位挂起寄存器位（见 *Section 4.3.4: Interrupt set-pending register x (NVIC_ISPRx)*），或向STIR（见 *Section 4.3.8: Software trigger interrupt register (NVIC_STIR)*）写入以使SGI挂起。

一个待处理的中断会保持待处理状态，直到以下其中一项发生：

- 处理器进入中断服务例程以处理中断。这将中断的状态从待处理变为激活。然后： 对于等级敏感中断，当处理器从中断服务例程返回时，NVIC 会采样中断信号。如果信号被激活，中断的状态将变为待处理，这可能导致处理器立即重新进入中断服务例程。否则，中断的状态将变为非激活。 对于脉冲中断，NVIC 会持续监控中断信号，如果该信号被触发，则中断的状态将变为待处理且激活。在这种情况下，当处理器从中断服务例程返回时，中断的状态变为待处理，这可能导致处理器立即重新进入中断服务例程。如果处理器在中断服务例程期间中断信号未被触发，则当处理器从中断服务例程返回时，中断的状态将变为非激活。

- 软件 向相应的中断清除挂起寄存器写入 {v*}。 特比特.
  对于电平敏感中断，如果中断信号仍保持激活状态，则中断状态保持不变。否则，中断状态将变为非激活状态。
  对于脉冲中断，中断状态变为：– 非活动状态，如果状态处于挂起状态
  – 活动，如果状态是活动且待处理的。

4.3.10 NVIC设计提示和技巧

确保软件使用正确对齐的寄存器访问。处理器不支持对NVIC寄存器的未对齐访问。请参阅各个寄存器的描述以了解支持的访问大小。

中断即使其被禁用也可以进入待处理状态。禁用中断仅阻止处理器处理该中断。

在将VTOR编程为重新定位向量表之前，请确保新向量表的向量表条目已为故障处理程序、非屏蔽中断以及所有已启用的异常中断设置。如需更多信息，请参见 *Section 4.4.4: Vector table offset register (VTOR) on page 227*。

NVIC 编程提示

软件使用CPSIE I和CPSID I指令来启用和禁用中断。CMSIS为这些指令提供了以下内联函数：

void __disable_irq(void) // 禁用中断void __enable_irq(void) // 启用中断

此外，C MSIS 提供了一系列用于 NVIC 控制的功能，包括:

表48. CMSIS对NVIC的控制函数

CMSIS interrupt control function	Description
void NVIC_SetPriorityGrouping(uint32_t priority_grouping)	Set the priority grouping
void NVIC_EnableIRQ(IRQn_t IRQn)	Enable IRQn
void NVIC_DisableIRQ(IRQn_t IRQn)	Disable IRQn
uint32_t NVIC_GetPendingIRQ (IRQn_t IRQn)	Return true (IRQ-Number) if IRQn is pending
void NVIC_SetPendingIRQ (IRQn_t IRQn)	Set IRQn pending
void NVIC_ClearPendingIRQ (IRQn_t IRQn)	Clear IRQn pending status
uint32_t NVIC_GetActive (IRQn_t IRQn)	Return the IRQ number of the active interrupt
void NVIC_SetPriority (IRQn_t IRQn, uint32_t priority)	Set priority for IRQn
uint32_t NVIC_GetPriority (IRQn_t IRQn)	Read priority of IRQn
void NVIC_SystemReset (void)	Reset the system

输入参数 IRQn 是 IRQ 编号，参见 *Table 17: Properties of the different exception types on page 38*。关于这些函数的更多信息，请参见 CMSIS 文档。

4.3.11 NVIC寄存器映射

此表显示NVIC寄存器映射及复位值。主NVIC寄存器块的基地址为0xE000E100。NVIC_STIR寄存器位于单独的块中，地址为0xE000EF00。

表49. NVIC寄存器映射及复位值

Offset	Register	13	03	92	82	72	62	52	42	32	22	12	02	91	81	71	61	51	41	31	21	11	01	9	8	7	6	5	4	3	2	1	0
0x100	NVIC_ISER0	SETENA[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x104	NVIC_ISER1	SETENA[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x11C	NVIC_ISER7	Reserved															SETENA [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x180	NVIC_ICER0	CLRENA[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x184	NVIC_ICER1	CLRENA[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x19C	NVIC_ICER7	Reserved															CLRENA [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x200	NVIC_ISPR0	SETPEND[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x204	NVIC_ISPR1	SETPEND[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x21C	NVIC_ISPR7	Reserved															SETPEND [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x280	NVIC_ICPR0	CLRPEND[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x284	NVIC_ICPR1	CLRPEND[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x29C	NVIC_ICPR7	Reserved															CLRPEND [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x300	NVIC_IABR0	ACTIVE[31:0]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

T表49. NVIC寄存器映射及复位值（续 尤德)

Offset	Register	13	03	92	82	72	62	52	42	32	22	12	02	91	81	71	61	51	41	31	21	11	01	9	8	7	6	5	4	3	2	1	0
0x304	NVIC_IABR1	ACTIVE[63:32]																															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x31C	NVIC_IABR7	Reserved															ACTIVE [239:224]																
	Reset Value	-	-	-	-	-	-	-	-								0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x400	NVIC_IPR0	IP[3]								IP[2]								IP[1]								IP[0]							
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x404	NVIC_IPR1	IP[7]								IP[6]								IP[5]								IP[4]							
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
:	:	:																															
0x4EC	NVIC_IPR59	IP[239]								IP[238]								IP[237]								IP[236]							
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
SCB registers																																	
Reserved																																	
0xE00	NVIC_STIR	Reserved																								INTID[8:0]							
	Reset Value																									0	0	0	0	0	0	0	0

4.4 系统控制块 (SCB)

System control block（SCB）提供系统实现信息和系统控制。这包括对系统异常的配置、控制和报告。

表 50. 系统控制块寄存器摘要

Address	Name	Type	Required privilege	Reset value	Description
0xE000E008	ACTLR	RW	Privileged	0x00000000	*Table 4.4.1: Auxiliary control register (ACTLR) on page 222*
0xE000ED00	CPUID	RO	Privileged	0x410FC241	*Table 4.4.2: CPUID base register (CPUID) on page 224*
0xE000ED04	ICSR	RW(1)	Privileged	0x00000000	*Table 4.4.3: Interrupt control and state register (ICSR) on page 225*
0xE000ED08	VTOR	RW	Privileged	0x00000000	*Table 4.4.4: Vector table offset register (VTOR) on page 227*
0xE000ED0C	AIRCR	RW(1)	Privileged	0xFA050000	*Table 4.4.5: Application interrupt and reset control register (AIRCR) on page 228*
0xE000ED10	SCR	RW	Privileged	0x00000000	*Table 4.4.6: System control register (SCR) on page 230*
0xE000ED14	CCR	RW	Privileged	0x00000200	*Table 4.4.7: Configuration and control register (CCR) on page 231*
0xE000ED18	SHPR1	RW	Privileged	0x00000000	*Table 4.4.8: System handler priority registers (SHPRx) on page 233*
0xE000ED1C	SHPR2	RW	Privileged	0x00000000	
0xE000ED20	SHPR3	RW	Privileged	0x00000000	
0xE000ED24	SHCSR	RW	Privileged	0x00000000	*Table 4.4.9: System handler control and state register (SHCSR) on page 235*
0xE000ED28	CFSR	RW	Privileged	0x00000000	*Table 4.4.10: Configurable fault status register (CFSR; UFSR+BFSR+MMFSR) on page 237*
0xE000ED28	MMSR (2)	RW	Privileged	0x00	MemManage Fault Status Register *Table 4.4.10 on page 237*
0xE000ED29	BFSR(2)	RW	Privileged	0x00	BusFault Status Register *Table 4.4.10 on page 237*
0xE000ED2A	UFSR(2)	RW	Privileged	0x0000	UsageFault Status Register *Table 4.4.10 on page 237*
0xE000ED2C	HFSR	RW	Privileged	0x00000000	*Table 4.4.14: Hard fault status register (HFSR) on page 241*
0xE000ED34	MMAR	RW	Privileged	Unknown	*Table 4.4.15: Memory management fault address register (MMFAR) on page 242*
0xE000ED38	BFAR	RW	Privileged	Unknown	*Table 4.4.16: Bus fault address register (BFAR) on page 242*
0xE000ED3C	AFSR	RW	Privileged	0x00000000	*Table 4.4.17: Auxiliary fault status register (AFSR) on page 243*

1. 请参阅寄存器描述以获取更多信息。2. CFSR的一个子寄存器。

4.4.1 辅助控制寄存器 (ACTLR)

地址偏移量：0x00（基地址 = 0xE000 E008）

复位值：0x0000 0000

所需权限：特权

默认情况下，该寄存器被设置为提供Cortex-M4处理器的最佳性能，并且通常不需要修改。ACTLR寄存器为以下处理器功能提供禁用位：

- 信息技术折叠
- 写缓冲区用于访问默认内存映射
- 多周期指令的中断

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						DISOOFP	DISFPCA						DISFOLD	DISDEFWBUF	DISMCYCINT
						rw	rw						rw	rw	rw

位31:10保留 位9 DISOOFP 禁用浮点指令相对于整数指令的乱序执行。

第8位 DISFPCA 禁用 CONTROL.FPCA 的自动更新。该位的值应写为零或保留（SBZP）。

位7:3保留

位 2 DISFOLD

禁用IT指令折叠：0：启用IT指令折叠。1：禁用IT指令折叠。

在某些情况下，处理器可以在执行IT指令的同时开始执行IT块中的第一条指令。这种行为称为IT折叠，可以提高性能。然而，IT折叠可能导致循环中的抖动。如果任务必须避免抖动，请在执行任务前将DISFOLD位设置为1，以禁用IT折叠。

位1 DISDEFWBUF

此位仅影响Cortex-M4处理器中实现的写缓冲区。禁用写缓冲区的使用，以在默认内存映射访问期间进行操作：这会导致所有BusFaults变为精确的BusFaults，但会降低性能，因为任何对内存的存储操作都必须在处理器执行下一条指令之前完成。

0: 启用写缓冲区使用
1: 禁用写缓冲区：存储到内存的操作会在下一条指令之前完成。

位 0 DISMCYCINT

禁用多周期指令的中断。当设置为1时，禁用{LDM}和{STM}指令的中断。这会增加处理器的中断延迟，因为处理器在压栈当前状态并进入中断处理程序之前，任何{LDM}或{STM}指令都必须完成。

0: 启用处理器的中断延迟（load/store和multiply/divide操作）。1: 禁用中断延迟。

4.4.2 CPUID基寄存器 (CPUID)

地址偏移量: 0x00

复位值：0x410F C241

所需权限：特权

CPUID寄存器包含处理器的部件号、版本和实现信息。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Implementer								Variant				Constant			
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PartNo												Revision			
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Bits 31:24 **Implementer**: Implementer code
0x41: Arm

Bits 23:20 **Variant**: Variant number
The r value in the r*n*p*n* product revision identifier
0x0: revision 0

Bits 19:16 **Constant:** Reads as 0xF

Bits 15:4 **PartNo**: Part number of the processor
0xC24: = Cortex-M4

Bits 3:0 **Revision**: Revision number
The p value in the r*n*p*n* product revision identifier, indicates patch release.
0x1: = patch 1

4.4.3 中断控制和状态寄存器 (ICSR)

地址偏移量：0x04

复位值：0x0000 0000

所需权限：特权

国际科学理事会:

- 提供：—— 为 *Non-Maskable Interrupt*（NMI）异常设置挂起位 —— 为 PendSV 和 SysTick 异常设置挂起和清除挂起位

- 表示:
  – 当前正在处理的异常编号 – 是否有被抢占的活动异常 – 最高优先级
  待处理异常的异常编号 – 是否有待处理的中断

注意：当您向ICSR写入时，效果不可预测，如果：

- 将 1 写入 PENDSVSET 位，并将 1 写入 PENDSVCLR 位
- 向PENDSTSET位写入1，并向PENDSTCLR位写入1。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NMIPENDSET	Reserved		PENDSVSET	PENDSVCLR	PENDSTSET	PENDSTCLR	Reserved		ISRPREEMPT	Reserved			VECTPENDING[6:4]		
rw			rw	w	rw	w			r				r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
VECTPENDING[3:0]				ISRPENDING	Reserved		VECTACTIVE[8:0]								
r	r	r	r	r			rw	rw	rw	rw	rw	rw	rw	rw	rw

第31位 NMIPENDSET：非屏蔽中断设置挂起位。
写入：0：无效果 1：将NMI异常状态更改为待处理。读取：0：NMI异常未待处理 1：NMI异常待处理 因为NMI是最高优先级的异常，通常处理器在检测到对该位写入1时会立即进入NMI异常处理程序，进入处理程序后会将该位清除为0。NMI异常处理程序读取该位时，仅当处理器在执行该处理程序期间重新发出NMI信号时，才会返回1。

Bits 30:29 Reserved

位28 PENDSVSET：PendSV 设置挂起位。写入：0：无影响 1：将 PendSV 异常状态改为挂起。读取：0：PendSV 异常未挂起 1：PendSV 异常已挂起 向该位写入 1 是将 PendSV 异常状态设置为挂起的唯一方式。位27 PENDSVCLR：PendSV 清除挂起位。该位仅支持写入。读取时，值未知。0：无影响 1：从 PendSV 异常中移除挂起状态。

Bit 26 PENDSTSET: SysTick异常设置为挂起位。写入：0
：无影响 1：将SysTick异常状态设置为挂起

阅读：
0: SysTick异常未挂起 1: SysTick异常已挂起

位25 PENDSTCLR：SysTick异常清除挂起位。只写。读取时值未知。0：无影响 1：从SysTick异常中移除挂起状态。

第24位保留，必须保持清除状态。

位23 该位保留用于调试，并且当处理器不在调试模式下时读为零。

位22 ISRPENDING: 中断挂起标志，排除NMI和故障。0: 中断未挂起 1: 中断挂起

位21:19保留，必须保持清除。

位18:12 {v*}：待处理向量。指示最高优先级待处理且启用的异常的异常号。0：无待处理异常 其他值：最高优先级待处理且启用的异常的异常号。该字段指示的值包含BASEPRI和FAULTMASK寄存器的影响，但不包含PRIMASK寄存器的任何影响。

位11 RETTOBASE：返回基线级别。表示是否存在被抢占的活动异常：0：存在需要执行的被抢占的活动异常 1：没有活动异常，或者当前正在执行的异常是唯一的活动异常。

位10:9 保留

位 8:0 VECTACTIVE 活动向量。包含当前活动的异常编号：0：线程模式 其他值：当前活动异常的异常编号 (1)。

Note: Subtract 16 from this value to obtain CMSIS IRQ number required to index into the Interrupt Clear-Enable, Set-Enable, Clear-Pending, Set-Pending, or Priority Registers, see Table 6 on page 22.

1. 这是 的 s 与IPSR的bits[8:0]相同，参见 *Interrupt program status register on page 22*。

4.4.4 向量表偏移量寄存器 (VTOR)

地址偏移量: 0x08 复位值: 0x0
000 0000 所需权限: 特权

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		TBLOFF[29:16]													
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TBLOFF[15:9]							Reserved								
rw	rw	rw	rw	rw	rw	rw									

位31:30 保留，必须保持清除

位29:9 TBLOFF：向量表基地址偏移字段。它包含从内存地址0x00000000到表基地址的偏移量的位[29:9]。设置TBLOFF时，必须将偏移量对齐到向量表中的异常条目数量。最小对齐要求为128个字。表对齐要求意味着表偏移量的位[8:0]始终为零。位29决定向量表是在代码还是SRAM内存区域。0：代码 1：SRAM
Note: Bit 29 is sometimes called the TBLBASE bit.

位 8:0 保留，必须保持清除

4.4.5 应用中断和复位控制寄存器 (AIRCR)

地址偏移量：0x0C

复位值：0xFA05 0000

所需权限：特权

AIRC 提供了异常模型的优先分组控制、数据访问的字节序状态以及系统的复位控制。

写入该寄存器时，必须将0x5FA写入VECTKEY字段，否则处理器会忽略写入操作。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
VECTKEYSTAT[15:0](read)/ VECTKEY[15:0](write)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SSENADNE	Reserved				PRIGROUP			Reserved					SYS RESET REQ	VECT CLR ACTIVE	VECT RESET
r					rw	rw	rw						w	w	w

位31:16 VECTKEYSTAT/ VECTKEY 寄存器密钥 读取为0xFA05 写入时，向VECTKEY写入0x5FA，否则写入将被忽略。位15 ENDIANESS 数据端序位 读取为0。0：小端序 位14:11 保留，必须保持清除 位10:8 PRIGROUP：中断优先级分组字段 该字段决定了组优先级与子优先级的划分，参见 *Binary point on page 228*。位7:3 保留，必须保持清除

位2 SYSRESETREQ 系统复位请求 该位用于强制对除调试外的所有主要组件进行大范围系统复位。该位读为0。 0: 无系统复位请求；1: 向外部系统发出请求复位的信号。

位1 VECTCLRACTIVE 保留用于调试。该位读为0。在向寄存器写入时，必须将该位写为0，否则行为不可预测。位0 VECTRESET 保留用于调试。该位读为0。在向寄存器写入时，必须将该位写为0，否则行为不可预测。

二进制小数点

PRIGROUP 字段指示将中断优先级寄存器中的 PRI_*n* 字段分割为独立的 *group priority* 和 *subpriority* 字段的二进制点位置。 *Table 51* 表示 PRIGROUP 值如何控制这种分割。如果您实现少于 8 个优先级

位您可能需要在这里进行更多解释，并希望从表格中删除无效行，并修改条目以调整列数
。

表格 51. 优先级分组

PRIGROUP [2:0]	Interrupt priority level value, PRI_*N*[7:4]			Number of	
	Binary point(1)	Group priority bits	Subpriority bits	Group priorities	Sub priorities
0b0xx	0bxxxx	[7:4]	None	16	None
0b100	0bxxx.y	[7:5]	[4]	8	2
0b101	0bxx.yy	[7:6]	[5:4]	4	4
0b110	0bx.yyy	[7]	[6:4]	2	8
0b111	0b.yyyy	None	[7:4]	None	16

1. PRI_*n*[7:4]字段显示二进制点。x表示组优先级字段位，y表示子优先级字段位。

Determining preemption of an exception uses only the group priority field, see *Section 2.3.6: Interrupt priority grouping on page 41*.

4.4.6 系统控制寄存器 (SCR)

地址偏移量: 0x10

复位值：0x0000 0000

所需权限：特权

SCR控制进入和退出低功耗状态的特性。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved											SEVONPEND	Res.	SLEEPDEEP	SLEEPONEXIT	Res.
											rw		rw	rw	

位31:5 保留，必须保持清零

位4 SEVEONPEND 在待处理位上发送事件

当一个事件或中断进入待处理状态时，事件信号会唤醒处理器脱离WFE。如果处理器未等待事件，则事件被注册并影响下一个WFE。

T处理器在执行SEV指令或外部事件0时也会唤醒：只有已启用的中断或事件才能唤醒处理器，禁用的中断是

排除1：已启用的事件和所有中断（包括被禁用的中断）都可以唤醒处理器。

第3位保留，必须保持清零

位 2 深度睡眠

控制处理器是否使用睡眠或深度睡眠作为其低功耗模式：0: 睡眠 1: 深度睡眠。

位1 {v*} 配置在从Handler模式返回到线程模式时的退出睡眠功能。将此位设置为1可使中断驱动的应用程序避免返回到空的主应用程序。0：返回线程模式时不睡眠。1：从中断服务例程返回时进入睡眠或深度睡眠。

第0位保留，必须保持清零

4.4.7 配置和控制寄存器 (CCR)

地址偏移量: 0x14

复位值：0x0000 0200

所需权限：特权

CCR控制进入Thread模式并启用：

- NMI、硬件故障以及被FAULTMASK升级的故障的处理程序用于忽略总线故障
- 捕获除以零和未对齐的访问
- 非特权软件访问STIR，参见*Software trigger interrupt register (NVIC_STIR) on page 216*。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						STK ALIGN	BFHF NMIGN	Reserved			DIV_0_ TRP	UN ALIGN_ TRP	Res.	USER SET MPEND	NON BASE THRD ENA
						rw	rw				rw	rw		rw	rw

位31:10 保留，必须保持清零

位 9 STKALIGN

配置异常入口的栈对齐。在异常入口时，处理器使用堆栈PSR的第9位来指示栈对齐。在从异常返回时，它使用该堆栈位来恢复正确的栈对齐。

0: 4字节对齐 1: 8
字节对齐

第8位 BFHFNMIGN

启用优先级为-1或-2的处理程序，使其忽略由加载和存储指令引起的总线数据故障。此功能适用于硬故障、NMI和FAULTMASK升级的处理程序。仅当处理程序及其数据位于绝对安全的内存中时，才将此位设置为1。此位的常规用途是探测系统设备和桥接器，以检测控制路径问题并进行修复。

0: 由加载和存储指令引起的数据显示故障会导致系统锁定 1: 在优先级为-1和-2下运行的处理程序忽略由加载和存储指令引起的数据显示故障。

位7:5 保留，必须保持清零

位4 DIV_0_TRP

启用故障或停止当处理器执行一个{v*}或{v*}指令，除数为0时：

0: 不要捕获除零错误 1:
捕获除零错误.

当此位被设置为0时，除以零会导致商为o f 0.

位3 UNALIGN_TRP 启用非对齐访问陷阱：0：不触发非对齐半字和字访问；1：触发非对齐半字和字访问。如果此位设置为1，非对齐访问将引发使用故障。非对齐的LDM、STM、LDRD和STRD指令始终引发故障，无论UNALIGN_TRP是否设置为1。

第2位保留，必须保持清零

位1 USERSETMPEND 允许无特权软件访问STIR，参见 *Software trigger interrupt register (NVIC_STIR) on page 216*：0：禁用 1：启用。

位0 NONBASETHRDENA 配置处理器如何进入线程模式。0：仅当没有活动的异常时，处理器才能进入线程模式。1：处理器可以在EXC_RETURN值的控制下从任何级别进入线程模式，参见 *Exception return on page 44*。

4.4.8 系统处理程序优先级寄存器 (SHPRx)

SHPR1-SHPR3 寄存器设置异常处理程序的优先级，0 到 255，这些处理程序具有可配置的优先级。

SHPR1-SHPR3 可字节访问。

系统故障处理程序及其每个处理程序的优先级字段和寄存器为:

表52. 系统故障处理程序优先级字段

Handler	Field	Register description
Memory management fault	PRI_4	*System handler priority register 1 (SHPR1)*
Bus fault	PRI_5	
Usage fault	PRI_6	
SVCall	PRI_11	*System handler priority register 2 (SHPR2) on page 233*
PendSV	PRI_14	*System handler priority register 3 (SHPR3) on page 234*
SysTick	PRI_15	

每个PRI_N字段为8位宽，但处理器仅实现每个字段的bits[7:3]部分，bits[3:0]在读取时视为零，并忽略写入操作（其中M=4）。

系统处理程序优先级寄存器1 (SHPR1)

地址偏移量：0x18

复位值：0x0000 0000

所需权限：特权

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								PRI_6[7:4]				PRI_6[3:0]			
								rw	rw	rw	rw	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PRI_5[7:4]				PRI_5[3:0]				PRI_4[7:4]				PRI_4[7:4]			
rw	rw	rw	rw	r	r	r	r	rw	rw	rw	rw	r	r	r	r

位31:24 保留，必须保持清除 位23:16 PRI_6：系统处理程序6的优先级，使用故障 位15:8 PRI_5：系统处理程序5的优先级，总线故障 位7:0 PRI_4：系统处理程序4的优先级，内存管理故障

系统处理程序优先级寄存器2 (SHPR2)

地址偏移量：0x1C 复位值：0
x0000 0000 所需权限：特权级

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRI_11[7:4]				PRI_11[3:0]				Reserved							
rw	rw	rw	rw	r	r	r	r								
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

位31:24 PRI_11：系统处理程序11的优先级，SVCall
位23:0 保留，必须保持清除

系统处理程序优先级寄存器3 (SHPR3)

地址: 0xE000 ED20　复位值: 0x0000 0000　所需特权: 特权

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PRI_15[7:4]				PRI_15[3:0]				PRI_14[7:4]				PRI_14[3:0]			
rw	rw	rw	rw	r	r	r	r	rw	rw	rw	rw	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留															

位31:24 PRI_15：系统处理程序15的优先级，SysTick异常 位23:16 PRI_14：系统处理程序14的优先级，PendSV 位15:0 保留，必须保持清除

4.4.9 系统处理程序控制和状态寄存器 (SHCSR)

地址偏移量：0x24

复位值：0x0000 0000

所需权限：特权

SHCSR 启用系统处理程序，并表示：

- 总线故障、内存管理故障和SVC异常的挂起状态
- 系统处理程序的活动状态

如果您禁用了系统处理程序且相应的故障发生，处理器将该故障视为硬件故障。{v*}

您可以向此寄存器写入以更改系统异常的待处理或激活状态。操作系统内核可以通过写入激活位执行上下文切换，从而更改当前异常类型。

- 软件在更改此寄存器中活动位的值时，若未正确调整堆栈内容，可能会导致处理器生成故障异常。确保写入此寄存器的软件保留当前活动状态并随后恢复。

- 在启用系统处理程序后，如果需要更改此寄存器中某位的值，必须使用读-修改-写操作以确保仅更改所需的位。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved													USG FAULT ENA	BUS FAULT ENA	MEM FAULT ENA
													rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SV CALL PEND ED	BUS FAULT PEND ED	MEM FAULT PEND ED	USG FAULT PEND ED	SYS TICK ACT	PEND SV ACT	Res.	MONIT OR ACT	SV CALL ACT	Reserved			USG FAULT ACT	Res.	BUS FAULT ACT	MEM FAULT ACT
rw	rw	rw	rw	rw	rw		rw	rw				rw		rw	rw

位31:19 保留位，必须保持清除 位18 USGFAULTENA：使用故障使能位，设置为1以使能 (1) 位17 BUSFAULTENA：总线故障使能位，设置为1以使能 (1) 位16 MEMFAULTENA：内存管理故障使能位，设置为1以使能 (1) 位15 SVCALLPENDED：SVC调用挂起位，当异常挂起时读为1 (2) 位14 BUSFAULTPENDED：总线故障异常挂起位，当异常挂起时读为1 (2) 位13 MEMFAULTPENDED：内存管理故障异常挂起位，当异常挂起时读为1 (2) 位12 USGFAULTPENDED：使用故障异常挂起位，当异常挂起时读为1 (2) 位11 SYSTICKACT：SysTick异常活动位，当异常处于活动状态时读为1 (3) 位10 PENDSVACT：PendSV异常活动位，当异常处于活动状态时读为1 位9 保留位，必须保持清除

Bit 8 MONITORACT: 调试监控器活动位，当调试监控器处于活动状态时读为1 Bit 7 SVC ALLACT: SVC调用活动位，当SVC调用处于活动状态时读为1 Bits 6:4 保留位，必须保持为0 Bit 3 USGFAULTACT: 使用故障异常活动位，当异常处于活动状态时读为1 Bit 2 保留位，必须保持为0 Bit 1 BUSFAULTACT: 总线故障异常活动位，当异常处于活动状态时读为1 Bit 0 MEMFAULTACT: 内存管理故障异常活动位，当异常处于活动状态时读为1

1. 使能位，设置为1以启用异常，或设置为0以禁用异常。2. 待处理位，如果异常处于待处理状态，则读取为1，否则读取为0。您可以向这些位写入以更改异常的待处理状态。3. 活动位，如果异常处于活动状态，则读取为1，否则读取为0。您可以向这些位写入以更改异常的活动状态，但请参见本节的注意事项。

4.4.10 配置 故障状态寄存器 (CFSR; UFSR+B FSR+MMFSR)

地址偏移量：0x28

复位值：0x0000 0000

所需权限：特权

以下子节描述组成CFSR的子寄存器：

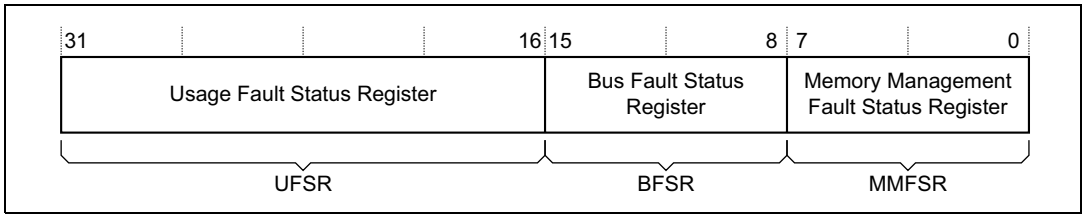
- *Usage fault status register (UFSR) on page 238*
- *Bus fault status register (BFSR) on page 239*
- *Memory management fault address register (MMFSR) on page 240*

{v*} 是字节可访问的。您可以按照以下方式访问 {v*} 或其子寄存器：

- 通过字访问0xE000ED28以访问完整的CFSR
- 通过字节访问读取MMFSR地址0xE000ED28
- 以半字访问方式访问MMFSR和BFSR，地址为0xE000ED28
- 以字节访问方式访问 BFSR 的 0xE000ED29 地址
- 以半字访问方式访问UFSR的0xE000ED2A地址。

CFSR 指示内存管理故障、总线故障或使用故障的原因。

图20. CFSR子寄存器



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved						DIVBY ZERO	UNALI GNED	Reserved				NOCP	INVPC	INV STATE	UNDEF INSTR
						rc_w1	rc_w1					rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BFARV ALID	Reserv ed	LSP ERR	STK ERR	UNSTK ERR	IMPRE CIS ERR	PRECI S ERR	IBUS ERR	MMAR VALID	Reserv ed	MLSP ERR	MSTK ERR	M UNSTK ERR	Res.	DACC VIOL	IACC VIOL
rw		rw	rw	rw	rw	rw	rw	rw		rw	rw	rw		rw	rw

位31:16 UFSR：参见 *Usage fault status register (UFSR) on page 238* 位15:8 BFSR：参见 *Bus fault status register (BFSR) on page 239* 位7:0 MMFSR：参见 *Memory management fault address register (MMFSR) on page 240*

4.4.11 使用故障状态寄存器 (UFSR)

位31:26 保留，必须保持清零

位25 DIVBYZERO：除以零使用故障。当处理器将此位设置为1时，用于异常返回的PC值指向导致除以零的指令。通过将CCR中的DIV_0_TRP位设置为1来启用除以零的陷阱，参见 *Configuration and control register (CCR) on page 231*。0：无除以零故障，或未启用除以零陷阱；1：处理器已执行了除数为0的SDIV或UDIV指令。

位24 UNALIGNED：未对齐访问使用故障。通过将条件码寄存器中的UNALIGN_TRP位设置为1来启用未对齐访问的陷阱，参见 *Configuration and control register (CCR) on page 231*。

未对齐的LDM、STM、LDRD和STRD指令始终会引发故障，无论UNALIGN_TRP的设置如何。0：无未对齐访问故障，或未启用未对齐访问陷阱；1：处理器已执行未对齐内存访问。

位23:20 保留，必须保持清零

位19 NOCP：无协处理器使用故障。处理器不支持协处理器指令：0：未因尝试访问协处理器而引发使用故障；1：处理器已尝试访问协处理器。

位18 INVPC：无效PC加载使用故障，由EXC_RETURN引发的无效PC加载造成：当该位设置为1时，异常返回时压入堆栈的PC值指向尝试执行非法PC加载的指令。0：无无效PC加载使用故障 1：处理器由于无效上下文或无效的EXC_RETURN值，尝试将EXC_RETURN非法加载到PC中。

位17 INVSTATE：无效状态使用故障。当该位设置为1时，用于异常返回的PC值被压栈，指向尝试非法使用EPSR的指令。如果未定义指令使用EPSR，则该位不会被设置为1。0：无无效状态使用故障 1：处理器尝试执行了非法使用EPSR的指令。

位16 UNDEFINSTR：未定义指令使用故障。当此位设置为1时，异常返回时压入堆栈的PC值指向未定义指令。未定义指令是指处理器无法解码的指令。0：无未定义指令使用故障；1：处理器尝试执行未定义指令。

4.4.12 总线故障状态寄存器 (BFSR)

位15 BFARVALID：*Bus Fault Address Register* (BFAR) 有效标志。处理器在地址已知的总线故障后将此位设置为1。其他故障可以将此位设置为0，例如稍后发生的内存管理故障。

如果发生总线故障并由于优先级升级为硬故障，硬故障处理程序必须将此位设置为0。这可以防止在返回到堆叠的活动总线故障处理程序时出现问题，此时BFAR值已被覆盖。

0: BFAR中的值不是一个有效的故障地址
1: BFAR 保存有效的故障地址。

第14位 保留，必须保持清除

位13 LSPERR：浮点延迟状态保存时发生总线故障。0：浮点延迟状态保存期间未发生总线故障。1：浮点延迟状态保存期间发生了总线故障 {v*}

位12 STKERR：异常入口时的堆栈总线故障。当处理器将此位设置为1时，SP仍会被调整，但堆栈上的上下文区域中的值可能不正确。处理器不会将故障地址写入BFAR。

0：无堆栈错误 1：异常入口的堆栈操作导致了一个或多个总线错误。

位11 UNSTKERR：异常返回时的出栈总线故障。此故障与处理程序相关联。这意味着当处理器将此位设置为1时，原始返回栈仍然存在。处理器不会从失败的返回调整SP，不会执行新的保存操作，也不会将故障地址写入BFAR。0：无出栈故障 1：异常返回时的出栈导致了一个或多个总线故障。

位10 IMPRECISERR：不精确的数据总线错误。当处理器将此位设置为1时，它不会将故障地址写入BFAR。这是一个异步故障。因此，如果在当前进程的优先级高于总线故障优先级时检测到该故障，总线故障将变为待处理状态，并且仅在处理器从所有更高优先级的进程返回后才会激活。如果在处理器进入不精确总线故障处理程序之前发生精确故障，处理程序会检测到IMPRECISERR被设置为1以及其中一个精确故障状态位也被设置为1。

0: 无不精确的数据总线错误
1: 一个数据总线错误已经发生，但是堆栈帧中的返回地址与导致错误的指令无关。

位9 PRECISERR：精确的数据总线错误。当处理器将此位设置为1时，它将故障地址写入BFAR。0：无精确的数据总线错误 1：发生数据总线错误，且用于异常返回的PC值指向导致故障的指令。{v*}

位8 IBUSERR：指令总线错误。处理器在预取指令时检测到指令总线错误，但只有在尝试执行出错的指令时，才会将IBUSERR标志设置为1。

当处理器将此位设置为1时，它不会将故障地址写入BFAR。
0: 无指令总线错误
1: 指令总线错误。

4.4.13 内存 内存管理故障地址寄存器 (MM FSR)

位7 MMARVALID：内存管理故障地址寄存器（MMAR）有效标志位。如果发生内存管理故障并由于优先级被提升为硬故障，硬故障处理程序必须将此位设置为0。这可以防止在返回到堆栈中的活动内存管理故障处理程序时出现问题，该处理程序的MMAR值已被覆盖。

0: MMAR中的值不是一个有效的故障地址 1:
MMAR保存一个有效的故障地址。

位6保留，必须保持清除

位5 MLSPERR:

0: 在浮点延迟状态保存过程中未发生内存管理故障 1: 在浮点延迟状态保存过程中发生了内存管理故障

位4 MSTKERR：异常入口堆栈操作时的内存管理故障。当此位为1时，SP仍会被调整，但堆栈上的上下文区域中的值可能不正确。处理器未将故障地址写入MMAR。

0: 无堆栈错误1: 异常入口的堆栈导致了一个或多个访问冲突。

位3 MUNSTKERR：异常返回时的栈展开错误。此错误会链接到处理程序。这意味着当该位为1时，原始返回栈仍然存在。处理器未从失败的返回调整SP，也未执行新的保存操作。处理器未将错误地址写入MMAR。

0: 无反向压栈错误1: 异常返回时的反向压栈导致一个或多个访问冲突。

第2位保留，必须保持清零

位1 DACCVIOL：数据访问违规标志。当该位为1时，异常返回时压入堆栈的PC值指向引发故障的指令。处理器已将尝试访问的地址加载到MMAR中。

0: 无数据访问违规故障1: 处理器尝试在不允许执行操作的位置进行加载或存储。

位1 IACCVIOL：指令访问违规标志。该故障会在任何对XN区域的访问时发生，即使内存保护单元被禁用或不存在。

当此位为1时，用于异常返回的栈值指向出错指令。处理器尚未将故障地址写入MMAR
。

0: 无指令访问违规故障 1: 处理器尝试从不允许执行的地址获取指令。

4.4.14 硬故障状态寄存器 (HFSR)

地址偏移量：0x2C

复位值：0x0000 0000

所需权限：特权

HFSR 提供有关触发硬故障处理程序的事件的信息。该寄存器可读可清除。这意味着寄存器中的位正常读取，但将任何位写为 1 会将其清零。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DEBUG_VT	FORCED	Reserved													
rc_w1	rc_w1														

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved														VECTTBL	Res.
														rc_w1	

位 31 DEBUG_VT：保留用于调试。在向寄存器写入时，必须将此位写为 0，否则行为不可预测。

位30 强制：强制硬故障。表示由无法处理的故障升级所生成的强制硬故障，该故障具有可配置的优先级，要么由于优先级不足，要么由于其被禁用。

当该位被设置为1时，硬故障处理程序必须读取其他故障状态寄存器以确定故障原因。0：无强制硬故障 1：强制硬故障。

位29:2 保留，必须保持清零

位1 VECTTBL：向量表硬故障。表示在异常处理过程中读取向量表时发生的总线故障。此错误始终由硬故障处理程序处理。当此位设置为1时，用于异常返回的PC值被压栈，指向被异常抢占的指令。0：无总线故障；1：有总线故障。

第0位保留，必须保持清零

4.4.15 内存 内存管理故障地址寄存器 (MM 联邦航空法规)

地址偏移量：0x34 复位值：
未定义 所需权限：特权级

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MMFAR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MMFAR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:0 MMFAR：内存管理故障地址 当MMFSR的MMARVALID位被设置为1时，该字段保存了生成内存管理故障的地址。当未对齐访问发生故障时，地址是发生故障的实际地址。由于单个读或写指令可以拆分为多个对齐访问，故障地址可以是请求的访问大小范围内的任何地址。MMFSR寄存器中的标志指示故障原因，以及MMFAR中的值是否有效。参见
Configurable fault status register (CFSR; UFSR+BFSR+MMFSR) on page 237。

4.4.16 总线故障地址寄存器（BFAR）

地址偏移量：0x38

复位值：未定义

所需权限：特权

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BFAR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BFAR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:0 BFAR: 总线故障地址 当BFSR中的BFARVALID位被设置为1时，该字段保存导致总线故障的地址。当未对齐的访问导致故障时，BFAR中的地址是指令请求的地址，即使该地址并非实际故障地址。BFSR寄存器中的标志位指示故障原因，并说明BFAR中的值是否有效。参见
Configurable fault status register (CFSR; UFSR+BFSR+MMFSR) on page 237。

4.4.17 辅助故障状态寄存器 (AFSR)

地址偏移量：0x3C 复位值：
未定义 所需权限：特权级

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
IMPDEF[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IMPDEF[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:0 实现定义的。AFSR 包含额外的系统故障信息。这些位映射到 AUXFAULT 输入信号。该寄存器为只读，写1清除。这意味着寄存器中的位正常读取，但将任何位写为1会将其清除为0。每个 AFSR 位直接映射到处理器的 AUXFAULT 输入，且输入上的单周期 HIGH 信号会将对应的 AFSR 位设置为1。它保持为1，直到您将该位写为1以清除为零。当 AFSR 位被锁存为1时，不会发生异常。如果需要异常，请使用中断。

4.4.18 系统控制块设计提示和技巧

确保软件使用正确大小的对齐访问来访问系统控制块寄存器：

- 除CFSR和SHPR1-SHPR3外，必须使用对齐的字访问
- 对于CFSR和SHPR1至SHPR3，可以使用字节、对齐的半字或字的访问方式。

处理器 doe 不支持对系统控制块的非对齐访问 ck寄存器
在故障处理程序中。以确定真实的故障地址：
1. 读取并保存 MMFAR 或 BFAR 值。2. 读取 MMFSR 中的 MMARVALID 位，或 BFSR 中的 BFARVALID 位。MMFAR 或 BFAR 地址仅当该位为 1 时才有效。

软件必须遵循此顺序，因为另一个更高优先级的异常可能会修改 MMFAR 或 BFAR 的值。例如，如果更高优先级的处理程序抢占当前的故障处理程序，另一个故障可能会修改 MMFAR 或 BFAR 的值。

4.4.19 SCB寄存器映射

该表展示了系统控制块寄存器映射及复位值。SCB寄存器块的基地址为0xE000 ED00，用于描述在*Table 53*中的寄存器。

表53. SCB寄存器映射及复位值

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	CPUID	Implementer								Variant				Constant				PartNo												Revision			
	Reset Value	0	1	0	0	0	0	0	1	0	0	0	1	1	1	1	1	1	1	0	0	0	0	1	0	0	0	1	1	0	0	0	1
0x04	ICSR	NMIPENDSET	Reserved		PENDSVSET	PENDSVCLR	PENDSTSET	PENDSTCLR	Reserved		ISRPENDING	VECTPENDING[9:0]										RETOBASE	Reserved		VECTACTIVE[8:0]								
	Reset Value	0			0	0	0	0			0	0	0	0	0	0	0	0	0	0	0	0			0	0	0	0	0	0	0	0	0
0x08	VTOR	Reserved		TABLEOFF[29:9]																					Reserved								
	Reset Value			0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0									
0x0C	AIRCR	VECTKEY[15:0]																ENDIANESS	Reserved				PRIGROUP[2:0]			Reserved					SYSRESETREQ	VECTCLRACTIVE	VECTRESET
	Reset Value	1	1	1	1	1	0	1	0	0	0	0	0	0	1	0	1	0					0	0	0						0	0	0
0x10	SCR	Reserved																											SEVONPEND	Reserved	SLEEPDEEP	SLEEPONEXIT	Reserved
	Reset Value																												0		0	0	
0x14	CCR	Reserved																						STKALIGN	BFHFNMIGN	Reserved			DIV_0_TRP	UNALIGN_TRP	Reserved	USERSETMPEND	NONBASETHRDENA
	Reset Value																							1	0				0	0		0	0
0x18	SHPR1	Reserved								PRI6								PRI5								PRI4							
	Reset Value									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

表53. SCB寄存器映射及复位值（续） 尤德)

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x1C	SHPR2	PRI11								Reserved																							
	Reset Value	0	0	0	0	0	0	0	0																								
0x20	SHPR3	PRI15								PRI14								Reserved															
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0																
0x24	SHCSR	Reserved																ANETLUAFGSU	ANETLUAFSUB	ANETLUAFMEM	DENNEPLLACVS	DENNEPTLUAFSUB	DENNEPTLUAFMEM	DENNEPTLUAFGSU	TCAKCTSYS	TCAVSDNEP	devreseR	TCAROTNOM	TCALLACVS	devreseR	TCATLUAFGSU	devreseR	TCATLUAFSUB	TCATLUAFMEM
	Reset Value																	0	0	0	0	0	0	0	0	0		0	0		0		0	0
0x28	CFSR	UFSR																BFSR								MMFSR							
	Reset Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x2C	HFSR	TVGUBED	DECROF	Reserved																											LBTTCEV	devreseR	
	Reset Value	0	0																												0		
0x34	MMAR	MMAR[31:0]																															
	Reset Value	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x38	BFAR	BFAR[31:0]																															
	Reset Value	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x3C	AFSR	IMPDEF[31:0]																															
	Reset Value	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

4.5 系统定时器 (STK)

处理器具有一个24位系统定时器SysTick，该定时器从装载值倒计数至零，在下一个时钟边沿时重新加载（回绕至）STK_LOAD寄存器中的值，然后在后续时钟中继续倒计数。

当处理器在调试时暂停，计数器不会递减。

表54. 系统定时器寄存器摘要

Address	Name	Type	Required privilege	Reset value	Description
0xE000E010	STK_CTRL	RW	Privileged	0x00000000	*SysTick control and status register (STK_CTRL) on page 247*
0xE000E014	STK_LOAD	RW	Privileged	Unknown	*SysTick reload value register (STK_LOAD) on page 248*
0xE000E018	STK_VAL	RW	Privileged	Unknown	*SysTick current value register (STK_VAL) on page 249*
0xE000E01C	STK_CALIB	RO	Privileged	0xC0000000	*SysTick calibration value register (STK_CALIB) on page 250*

4.5.1 系统定时器控制和状态寄存器 (STK_C TRL)

地址偏移量：0x00 复位值：0x0000 0000 所需权限：特权 S
ysTick CTRL寄存器启用了SysTick功能。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															COUNT FLAG
															rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													CLKSO URCE	TICK INT	EN ABLE
													rw	rw	rw

位31:17 保留位，必须保持为0。位16 COUNTFLAG：自上次读取此位以来，如果计时器计数到0，则返回1。位15:3 保留位，必须保持为0。位2 CLKSOURCE：时钟源选择 选择时钟源。0：AHB/8 1：处理器时钟（AHB）位1 TICKINT：SysTick异常请求使能 0：计数到零时不触发SysTick异常请求 1：计数到零时触发SysTick异常请求。

Note: Software can use COUNTFLAG to determine if SysTick has ever counted to zero. 位0 ENABLE：计数器使能 使能计数器。当ENABLE设置为1时，计数器从

加载寄存器，然后开始倒计时。当计数达到0时，将COUNTFLAG设置为1，并根据TICKINT的值可选地触发SysTick。然后加载RELOAD。

重新设置值，开始计数。0：计数

器禁用 1：计数器启用

4.5.2 版本 SysTick重载值寄存器 (STK_LOAD)

地址偏移量：0x04 复位值：0
x0000 0000 所需特权：特权

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								RELOAD[23:16]							
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RELOAD[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:24 保留，必须保持清除。

位23:0 RELOAD: RELOAD值

当计数器被启用且达到0时，LOAD寄存器指定要加载到STK_VAL寄存器中的起始值。

计算 RELOAD 值

重载值可以是0x00000001-0x00FFFFFF范围内的任意值。起始值为0是可能的，但没有效果，因为在从1计数到0时，SysTick异常请求和COUNTFLAG会被激活。

RELOAD值根据其用途进行计算：{v*}

要生成一个周期为N个处理器时钟周期的多拍定时器，请使用N-1的重载值。例如，如果需要每100个时钟脉冲触发一次SysTick中断，请将重载值设置为99。要延迟N个处理器时钟周期后触发一次SysTick中断，请使用N的重载值。例如，如果需要在100个时钟脉冲后触发SysTick中断，请将重载值设置为99。

4.5.3 SysTick当前值寄存器 (STK_VAL)

地址偏移量: 0x08 复位值: 0x0
000 0000 所需权限: 特权

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								CURRENT[23:16]							
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CURRENT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:24 保留，必须保持清零。位23:0 当前：当前计数器值 VAL寄存器包含SysTick计数器的当前值。读取操作将返回SysTick计数器的当前值。向VAL寄存器写入任何值都会将该字段清零，并将STK_CTRL寄存器中的COUNTFLAG位清零。

4.5.4 SysTick 校准值寄存器 (STK_CA 库)

地址偏移量：0x0C 复位值：0x0000000 所需权限：特权模式 CA
LIB寄存器指示SysTick校准属性。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
NO REF	SKEW	Reserved						TENMS[23:16]							
r	r							r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TENMS[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31 NOREF：NOREF标志。读取时为零。表示提供了独立的参考时钟。该时钟的频率为HCLK/8。

位30 SKEW：SKEW标志：指示TENMS值是否精确。读为1。由于TENMS未知，1毫秒不精确定时的校准值无法确定。这可能会影响SysTick作为软件实时时钟的适用性。

位29:24 保留，必须保持为清零状态。

位23:0 TENMS[23:0]: 校准值。表示当SysTick计数器以HCLK最大频率/8作为外部时钟运行时的校准值。该值与产品相关，请参阅产品参考手册中的SysTick校准值部分。当HCLK被设置为最大频率时，SysTick周期为1ms。如果校准信息未知，请根据处理器时钟或外部时钟的频率计算所需的校准值。

4.5.5 SysTick设计提示和技巧

SysTick计数器基于处理器时钟运行。如果该时钟信号因低功耗模式而停止，SysTick计数器也会停止。

确保软件使用对齐的字访问来访问SysTick寄存器。

SysTick计数器的重载值和当前值在复位时未定义，SysTick计数器的正确初始化序列是：

1. 程序重载值。 2. 清除当前值。 3. 程序
控制和状态寄存器。

4.5.6 SysTick寄存器映射

提供的表格显示了SysTick寄存器映射及其复位值。SysTick寄存器块的基地址为0xE000 E010
。

表55. SysTick寄存器映射以及复位值

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	STK_CTRL	Reserved																COUNTFLAG	Reserved													CLKSOURCE	TICKINT	ENABLE
	Reset Value																	0														1	0	0
0x04	STK_LOAD	Reserved																RELOAD[23:0]															
	Reset Value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	STK_VAL	Reserved																CURRENT[23:0]															
	Reset Value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	STK_CALIB	Reserved																TENMS[23:0]															
	Reset Value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

4.6 浮点运算单元（FPU）

Cortex-M4F FPU 支持 FPv4-SP 浮点扩展。

浮点运算单元完全支持单精度的加法、减法、乘法、除法、乘加和平方根运算。它还提供固定点和浮点数据格式之间的转换，以及浮点常量指令。{v*}

浮点运算单元提供符合ANSI/IEEE标准754-2008的浮点运算功能，该标准也称为IEEE 754浮点数运算标准。

浮点处理器包含32个单精度扩展寄存器，您也可以将其作为16个双字寄存器进行加载、存储和移动操作。

*Table 56*显示了Cortex-M4F系统控制块 (SCB)中的浮点系统寄存器。FP扩展的附加寄存器的基地址为0xE000 ED00。

表56. Cortex-M4F浮点系统寄存器

Address	Name	Type	Reset	Description
0xE000ED88	CPACR	RW	0x00000000	*Section 4.6.1: Coprocessor access control register (CPACR) on page 253*
0xE000EF34	FPCCR	RW	0xC0000000	*Section 4.6.2: Floating-point context control register (FPCCR) on page 253*
0xE000EF38	FPCAR	RW	-	*Section 4.6.3: Floating-point context address register (FPCAR) on page 255*
0xE000EF3C	FPDSCR	RW	0x00000000	*Section 4.6.5: Floating-point default status control register (FPDSCR) on page 257*
-	FPSCR	RW	-	*Section 4.6.4: Floating-point status control register (FPSCR) on page 255*

以下部分描述了浮点系统寄存器的实现，该实现特定于此处理器。

Note: For more details on the IEEE standard and floating-point arithmetic (IEEE 754), refer to the AN4044 Application note. Available from website www.st.com.

4.6.1 协处理器访问控制寄存器（CPACR）

地址偏移量（来自SCB）：0x88 复位值：0x0000000 所需特权：特权 CPACR寄存器指定协处理器的访问特权。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								CP11		CP10		Reserved			
								rw		rw					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

位31:24保留。读取时为零，写入时忽略。位23:20 CPn: [2n+1:2n] 对于n的值10和11。对协处理器n的访问权限。可能的

每个字段的值为：0b00：访问被拒绝。任何尝试访问都会生成NOCP使用错误。0b01：仅允许特权访问。非特权访问会生成NOCP错误。0b10：保留。任何访问的结果都是不可预测的。0b11：完全访问。位19:0保留。读取时视为零，写入则忽略。

4.6.2 浮点上下文控制寄存器（FPCCR）

地址偏移量：0x04 复位值0xC000000 所需权限：特权 FPCCR寄存器设置或返回FPU控制数据。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ASPEN	LSPEN	Reserved													
rw	rw														
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved							YDRNOM	dey'seR	BFRDY	YDRMM	HFRDY	DAERHT	dey'seR	USER	TCAPSL
							rw		rw	rw	rw	rw		rw	rw

位31 ASPEN：在执行浮点指令时启用CONTROL<2>设置。这会导致在浮点上下文中自动保存和恢复硬件状态，on
异常进入和退出。 0：在执行浮点指令时禁用 CONTROL<2> 设置。 1：在执行浮点指令时启用 CONTROL<2> 设置。

位 30 LSPEN：
0: 禁用浮点数上下文的自动延迟状态保存。1: 启用浮点数上下文的自动延迟状态保存。

位 29:9 保留。

第8位 MONRDY:
0: DebugMonitor 已禁用或优先级不允许设置 MON_PEND，当浮点栈帧被分配时。1: DebugMonitor 已启用且优先级允许设置 MON_PEND，当浮点栈帧被分配时。

位7 保留。

第6位 BFRDY:
0: BusFault 被禁用，或优先级不允许将 BusFault 处理程序设置为待处理状态，当分配浮点栈帧时。1: BusFault 已启用且优先级允许将 BusFault 处理程序设置为待处理状态

当浮点栈帧被分配时。

第5位 MMRDY:
0: MemManage 被禁用，或优先级不允许将 MemManage 处理程序设置为待处理状态，当浮点堆栈帧被分配时。1: MemManage 已启用，且优先级允许将 MemManage 处理程序设置为待处理状态，当浮点堆栈帧被分配时。

位 4 HFRDY:
0: 优先级不允许在浮点堆栈帧被分配时将HardFault处理程序设置为待处理状态。1: 优先级允许在浮点堆栈帧被分配时将HardFault处理程序设置为待处理状态。

第3位线程:
0: 当浮点栈帧被分配时，模式不是线程模式。1: 当浮点栈帧被分配时，模式是线程模式。

第2位保留。

位 1 用户:
0: 浮点栈帧分配时特权级别不是用户。1: 浮点栈帧分配时特权级别是用户。

第1位 LSPACT:
0: 惰性状态保存未激活。1: 惰性状态保存已激活。浮点栈帧已分配，但保存状态到该栈帧被延迟。

4.6.3 浮点上下文地址寄存器 (FP 汽车)

地址偏移量: 0x08 复位值: 0x0
000000 所需特权: 特权模式

FPCAR寄存器保存在异常栈帧上分配的未初始化浮点寄存器空间的位置。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADDRESS[31:16]															
rw															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDRESS[15:3]													Reserved		
rw															

位31:3 地址：指向在异常栈帧中分配的未分配浮点寄存器空间的位置。

4

.6.4

位2:0保留。读取时为零，写入时忽略。

浮点状态控制寄存器 (FPSC R)

地址偏移：未映射 复位值0x0000000 所需权限：特权级SCR寄存器提供浮点系统的所有必要用户级控制。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
N	Z	C	V	Reserved	AHP	DN	FZ	RMode		Reserved					
rw	rw	rw	rw		rw	rw	rw	rw	rw						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								IDC	Reserved		IXC	UFC	OFC	DZC	IOC
								rw			rw	rw	rw	rw	rw

位31 N：负条件码标志。浮点比较操作会更新这些标志。如需了解结果的更多细节，请参阅 *Table 57* 。0：操作结果为正、零、大于或等于。1：操作结果为负或小于。

位30 Z：零状态标志位。浮点比较操作会更新这些标志位。如需了解结果的更多细节，请参阅 *Table 57* 。0：操作结果不为零。1：操作结果为零。

位29 C：进位条件码标志。浮点比较操作会更新这些标志。如需更多结果详情，请参阅 *Table 57*。0：加法操作未产生进位位或减法操作产生借位位。1：加法操作产生进位位或减法操作未产生借位位。

位28 V：溢出条件代码标志。浮点比较操作会更新此标志。For mor e
结果详情，请参见 *Table 57*。0：未发生溢出 1：发生溢出。位27 保留。位26 AHP：替代半精度控制位：0：选择IEEE半精度格式。1：选择替代半精度格式。位25 DN：默认NaN模式控制位：0：NaN操作数传递到浮点运算的输出。1：任何涉及一个或多个NaN的操作返回默认NaN。位24 FZ：Flush-to-zero模式控制位：0：Flush-to-zero模式禁用。浮点系统的运行完全符合

IEEE 754标准。1: 冲零模式启用
。

B其 23:22 RMode: 舍入模式控制字段。指定的舍入模式被几乎所有浮点指令使用：0b00：就近舍入（RN）模式 0b01：向正无穷舍入（RP）模式 0b10：向负无穷舍入（RM）模式 0b11：向零舍入（RZ）模式。位 21:8 保留。位 7 IDC：输入非规范数累积异常位。浮点异常的累积异常位。1：表示自上次写入 0 以来该对应异常已发生。位 6:5 保留 位 4 IXC：不精确累积异常位。浮点异常的累积异常位。1：表示自上次写入 0 以来该对应异常已发生。位 3 UFC：下溢累积异常位。浮点异常的累积异常位

。

1：表示自上次写入0以来，相应的异常已发生。 位2 OFC：溢出累积异常位。浮点异常的累积异常位。 1：表示自上次写入0以来，相应的异常已发生。 位1 DZC：除以零累积异常位。浮点异常的累积异常位。 1：表示自上次写入0以来，相应的异常已发生。

Bit 0 IOC: 无效操作累积异常位。用于浮点异常的累积异常位。1: 表示自上次写入该位为0以来，相应的异常已发生。

表57. 浮点数比较对状态标志的影响

Comparison result	N	Z	C	V
Equal	0	1	1	0
Less than	1	0	0	0
Greater than	0	0	1	0
Unordered	0	0	1	1

4.6.5 弗洛 触发点默认状态控制寄存器 (F PDSCR)

地址偏移量：0x0C 复位值：0x0000000 所需特权：特权 FPDSCR寄存器保存浮点状态控制数据的默认值。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					AHP	DN	FZ	RMode		Reserved					
					rw	rw	rw	rw	rw						
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															

位 31:27 保留，必须保持清除。位 26 AHP：FPSCR.AHP 的默认值 位 25 DN：FPSCR.DN 的默认值 位 24 FZ：FPSCR.FZ 的默认值 位 23:22 RMode：FPSCR.RMode 的默认值 位 21:0 保留，必须保持清除。

4.6.6 启用FPU

FPU在复位后被禁用。您必须在使用任何浮点指令之前启用它。

示例展示了在特权模式和用户模式下启用FPU的代码序列。处理器必须处于特权模式才能读取和写入CPACR。

示例

```
                    ; CPACR 位于地址 0xE000ED88 LDR.W   R0, =0xE000ED88 ; 读取 CPAC
R LDR     R1, [R0] ; 将位 20-23 设置为启用 CP10 和 CP11 协处理器 ORR     R1, R1, #(0xF << 20) ; 将
修改后的值写回 CPACR STR     R1, [R0]; 等待存储完成 DSB ; 重置流水线，现在 FPU 已启用 ISB
```

4.6.7 启用和清除FPU异常interr 优普茨 {v*}

FPU异常标志通过中断控制器产生中断。FPU中断通过中断控制器进行全局控制。

系统配置控制器（SYSCFG）中还提供了一个屏蔽位，允许单独使能或禁用每个浮点运算单元标志的中断生成。

Note: In STM32F4xx devices there is no individual mask and the enable/disable of the FPU interrupts is done at interrupt controller level. As it occurs very frequently, the IXC exception flag is not connected to the interrupt controller in these devices , and cannot generate an interrupt. If needed, it must be managed by polling.

清除FPU异常标志取决于FPU上下文保存/恢复配置：

- 不保存浮点寄存器：当浮点上下文控制寄存器 (FPCCR) 位30 LSPEN=0 和位31 ASPEN=0 . 。您必须清除浮点状态和控制寄存器 (FPSCR) 中的中断源。 示例：register uint32_t fpscr_val = 0; fpscr_val = __get_FPSCR(); { 检查异常标志 } fpscr_val &= (uint32_t)~0x8F; // 清除所有异常标志 __set_FPSCR(fpscr_val);

- 延迟保存/恢复：当浮点上下文控制寄存器（FPCCR） 的第30位 LSPEN=1 和第31位 ASPEN=X. 在延迟浮点上下文保存/恢复的情况下，应向浮点状态和控制寄存器（FPSCR）进行虚拟读访问，以强制保存状态并清除FPSCR。 然后在栈中处理FPSCR。 示例：寄存器 uint32_t fpscr_val = 0; 寄存器 uint32_t reg_val = 0; reg_val = __get_FPSCR(); //虚拟访问 fpscr_val=*(__IO uint32_t*)(FPU->FPCAR +0x40); { 检查异常标志 } fpscr_val &= (uint32_t)~0x8F ; // 清除所有异常标志 *(__IO uint32_t*)(FPU->FPCAR +0x40)=fpscr_val; __DMB() ;

- 自动浮点寄存器保存/恢复：当浮点上下文控制寄存器（FPCCR） 位30 LSPEN= 为0，位31 ASPEN= 为1时。 在自动浮点上下文保存/恢复的情况下，应读取浮点状态和控制寄存器（FPSCR）以强制清除。 然后在堆栈中处理FPSCR。 示例：// FPU异常处理程序 void FPU_ExceptionHandler(uint32_t lr, uint32_t sp) {register uint32_t fpscr_val; if(lr == 0xFFFFFFE9) {

sp = sp + 0x60; }else if(lr == 0xFFFFFFED) { sp = __get_PSP() + 0x60 ; } fpscr_val = *(uint32_t*)sp; { check exception flags } fpscr_val &= (uint32_t)~0x8F ; // Clear all exception flags *(uint32_t*)sp = fpscr_val; __DMB() ; }// FPU IRQ Handler void __asm FPU_IRQHandler(void) {IMPORT FPU_ExceptionHandler MOV R0, LR // 将LR移动到R0 MOV R1, SP // 将SP保存到R1以避免FPU_ExceptionHandler对堆栈指针的任何修改 VMRS R2, FPSCR // dummy read access, 以强制清除 B FPU_ExceptionHandler BX LR }

5 修订历史

表 58. 文档修订历史

Date	Revision	Changes
20-Feb-2012	1	Initial release.
09-Jul-2012	2	Changed reset value in *Section 4.6.2: Floating-point context control register (FPCCR)*. Added *Table 1: Applicable products*.
04-Sep-2012	3	Added information on the STM32F3xxx Cortex-M4 processor. Added extra part numbers to *Table 1: Applicable products*. Added related documentation references to *Introduction*. Changed “IEEE754-compliant single-precision FPU” bullet in *Section 1.3.3: Cortex-M4 processor features and benefits summary*. Added information on extended interrupt/event controller to *Section 2.5.3: External event input / extended interrupt and event input*. Changed first “interrupt” bullet in *Section 4.3: Nested vectored interrupt controller (NVIC)*. Removed outdated reset value information in *Section 4.4.7: Configuration and control register (CCR)*, and for 0x14 offset in *Table 52: System fault handler priority fields*. Added a note about IEEE 754 to *Section 4.6: Floating point unit (FPU)*.
12-May-2014	4	Updated *Reference documents*. Updated *Section 4.4.1: Auxiliary control register (ACTLR)*. Updated *Section 4.5.1: SysTick control and status register (STK_CTRL)*.
18-Apr-2016	5	Updated: – *Introduction* – *Reference documents* – *Section 2.5.3: External event input / extended interrupt and event input* – *Section 4.6.7: Enabling and clearing FPU exception interrupts* – *Table 51: Priority grouping* Removed: – *Table 1: Applicable products*
02-Oct-2017	6	Updated document scope to include STM32L4+ Series impacting only the document’s title and cover page. Updated *Table 49: NVIC register map and reset values*
21-Feb-2019	7	Updated: – Document scope to include STM32MP1 Series, STM32WB Series, STM32G4 Series – Title and cover page – *Section 1: About this document* – General update of *Section 4.3: Nested vectored interrupt controller (NVIC)*

表 58. 文档修订历史（续）

Date	Revision	Changes
24-Jun-2019	8	Updated the *Introduction* and *Reference documents* to include the support for STM32H7 Series.
18-Dec-2019	9	Added STM32WL Series. Replaced SHCRS by SHCSR in *Table 50: Summary of the system control block registers* and *Table 53: SCB register map and reset values*.
23-Mar-2020	10	Replaced STM32H7 Series by STM32H745/755 and STM32H747/757 Lines, since Arm Cortex-M4 core is only present in these product lines.