C29x CPU

Reference Guide

Literature Number: ZHCUCH3A
NOVEMBER 2024 – REVISED MARCH 2025

目录

插图清单

表格清单

关于本手册

本手册介绍了 CPU 架构、中断、寻址模式、CPU 的功能安全和信息安全方面。此手册还描述了这些器件上可用的仿真功能。各章汇总如下。

结构概述

本章介绍了位于每个 F29x 器件核心的 CPU。本章包含一个存储器映射以及对将内核与存储器和外设连接起来的存储器接口的高级描述。

中央处理单元

本章对 CPU 的架构、寄存器和主要功能进行了说明。本章包含最重要 CPU 寄存器、状态寄存器 ISTS、DSTS 和 ESTS 中标志和控制位的详细说明。

CPU 中断架构概述

本章介绍了中断以及 CPU 如何处理中断。本章还阐释了复位对 CPU 的影响，并且包含对在为中断提供服务前由 CPU 执行的自动背景保存的讨论。

寻址模式

本章介绍了汇编语言指令接受数据以及访问寄存器和存储器位置的模式。本章介绍了如何在操作码中对寻址模式信息进行编码。

功能安全和信息安全单元

本章介绍了 F29x 架构采用的功能安全和信息安全方法。本章通过示例介绍任务隔离、链接、栈和区域的概念。

仿真特性

本章介绍了只可与一个 JTAG 端口和两个额外仿真引脚一同使用的 F29x 仿真特性。

德州仪器 (TI) 相关文档

有关这些器件的相关文档和开发支持工具的完整列表，请访问德州仪器 (TI) 网站 http://www.ti.com.cn。

术语表

TI 术语表　　本术语表列出并解释了术语、首字母缩略词和定义。

支持资源

TI E2E™ 中文支持论坛是工程师的重要参考资料，可直接从专家处获得快速、经过验证的解答和设计帮助。搜索现有解答或提出自己的问题，获得所需的快速设计帮助。

链接的内容由各个贡献者“按原样”提供。这些内容并不构成 TI 技术规范，并且不一定反映 TI 的观点；请参阅 TI 的使用条款。

商标

TI E2E™ is a trademark of Texas Instruments.

所有商标均为其各自所有者的财产。

章节 1

架构概述

C29x 是 C2000 系列中的浮点 CPU。本章对 CPU 的架构结构和组成部分进行了概述。

1.1 CPU 简介

C29x CPU 采用 VLIW（超长指令字）架构，并配备全面保护式流水线。C29x CPU 支持多种指令大小（16/32/48 位），指令包大小可变，可以包含多条并行执行的指令。例如，C29x CPU 架构能够并行执行多达 8 条指令。这由 CPU 内可以同时执行的多个功能单元实现。工作寄存器总共有 64 个，分为三个不同类别（Ax、Dx 和 Mx 寄存器组），用于支持 CPU 中的并行操作。除了工作寄存器外，CPU 还包含多个状态寄存器（DSTS、EST 和 ISTS），用于维护与执行和中断上下文相关的不同信息。

1.2 数据类型

C29x CPU 支持在存储器中使用以下数据类型：

支持 8、16、32、64 个数据类型：CPU 支持 8 位、16 位、32 位和 64 位运算。CPU 可以在单个运算（周期）中读写存储器大小为 8 位、16 位、32 位和 64 位的数据。

小端字节序格式：所有数据和寄存器均使用小端字节序格式。

数据对齐到字大小边界：16 位访问需要与 16 位字边界对齐（地址线 0 = 0）。32 位访问需要与 32 位字边界对齐（地址线 1,0 = 0,0）。64 位访问需要与 64 位字边界对齐（地址线 2,1,0 = 0,0,0）。

32 位和 64 位浮点：C29x CPU 支持使用 IEEE 格式的 32 位和 64 位浮点运算。这些值可以在定点和浮点寄存器之间移动，而不会产生存储器停滞。

C 编译器数据类型兼容性：

尺寸	C29x CPU 数据类型定义
char	8 位
short	16 位
int	32 位
long	32 位
long long	64 位
float	32 位
double	64 位
long double	64 位
指针	32 位

1.3 C29x CPU 系统架构

C29x CPU 系统架构由以下主要功能块组成，如图 1-1 所示。

- **C29x CPU 内核：**负责生成数据存储器和程序存储器地址；解码和执行指令；执行算术、逻辑和移位运算；控制 CPU 寄存器、数据存储器和程序存储器之间的数据传输
- **CPU 栈：**在安全环境中管理软件、受保护的调用和实时中断栈。
- **CPU 接口总线：**用于连接存储器和外设、时钟及控制 CPU 和仿真逻辑的信号
- **功能安全和信息安全单元 (SSU)：**在硬件中将安全性、存储器管理 (MPU) 和安全性作为一项功能来实现。
- **外设中断优先级扩展 (PIPE)：**管理所有外设中断源并确定其优先级。有关 PIPE 的更多详细信息，请参阅 *F29H85x 和 F29P58x 实时微控制器技术参考手册*。
- **C29x CPU 调试接口：**用于监控和控制 MCU 的各种器件和功能，以及测试器件运行。与 CPU 外部的调试子系统 (DebugSS) 和嵌入式实时分析和诊断 (ERAD) 单元连接。

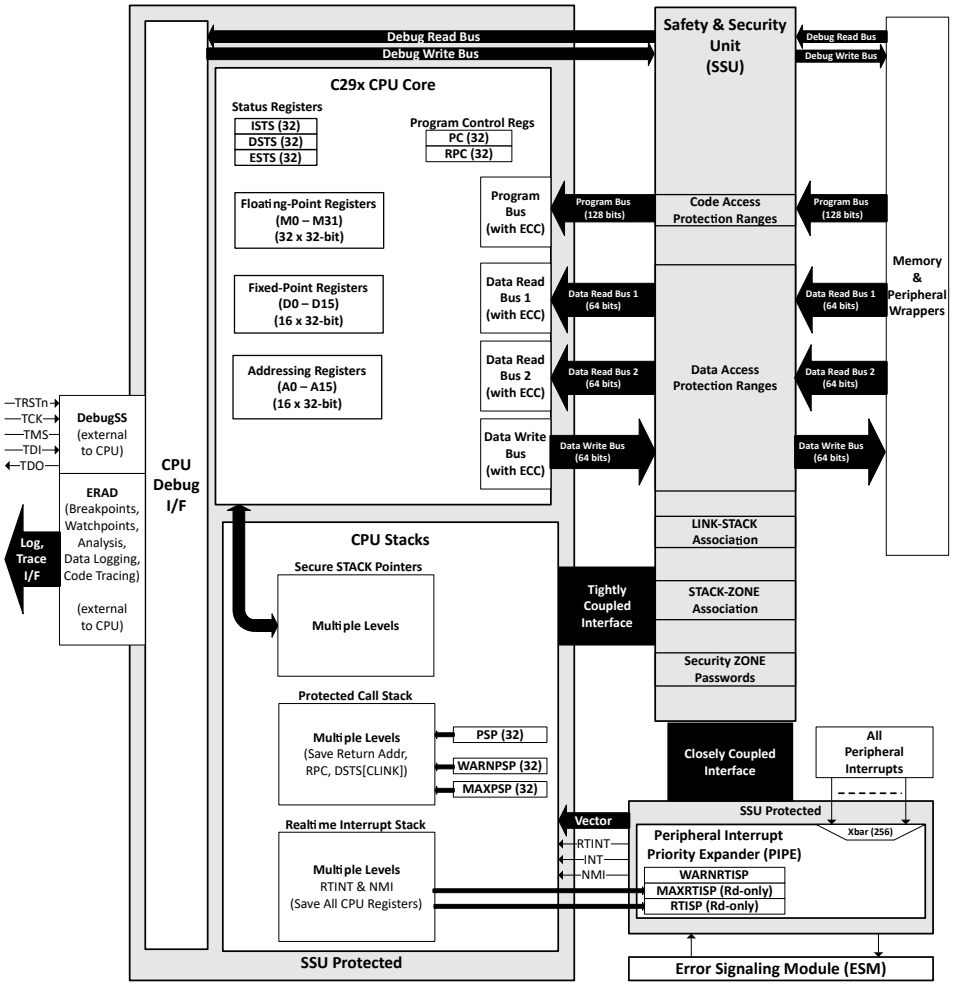

图 1-1. C29x CPU 系统架构

1.3.1 仿真逻辑

仿真逻辑包含以下特性。有关这些特性的更多详细信息，请参阅章节 6。

1. **代码执行控制**
   - 下载代码的能力
   - 支持断点/观察点
   - 运行、中止、单步执行
2. **系统可见性**
   - 访问系统存储器
   - 访问外设存储器
   - 在不中止 CPU 的情况下实时访问存储器/外设
3. **交叉触发**
   - 将事件从一个 CPU 子系统映射到另一个 CPU 子系统
   - 事件操作在相应的 CPU 子系统中处理
4. **安全性**
   - 根据器件的安全架构，访问从调试器到 HSM 模块/SSU 的安全质询响应路径
5. **性能分析**
   - C29x CPU 代码分析是使用 ERAD 完成的
6. **迹线**
   - C29x CPU 跟踪使用 ERAD PC 不连续跟踪完成
7. **复位**
   - 支持使用调试器进行 CPU 和系统复位

1.3.2 CPU 接口总线

C29x CPU 内核通过以下总线访问代码、数据和外设资源：

程序总线：程序总线用于从存储器子系统获取指令。程序总线的数据总线宽度为 128 位。该总线可以在单个周期中获取 128 位。C29x CPU 支持从 16 位到高达 128 位的指令数据包。每次获取指令都受 ECC 保护。

数据读取总线 1：数据读取总线 1 用于从存储器子系统或外设读取数据。数据读取总线 1 的数据总线宽度为 64 位。该总线在单周期内可以读取 8 位、16 位、32 位和 64 位数据。C29x CPU 上有两条数据读取总线。如果地址位于不同的物理存储器组中，则可以使用这些总线同时访问来自存储器的数据（请参阅器件特定数据表来识别物理存储器组）。如果同时访问同一存储块，则可以按任何顺序对访问进行仲裁或提供服务。有关物理组的详细信息，请参阅器件特定数据手册。数据读取总线 1 受 ECC 保护。

数据读取总线 2：数据读取总线 2 用于从存储器子系统或外设读取数据。数据读取总线 2 的数据总线宽度为 64 位。该总线在单周期内可以读取 8 位、16 位、32 位和 64 位数据。数据读取总线 2 受 ECC 保护。

数据写入总线：数据写入总线用于向存储器子系统或外设写入数据。数据写入总线的数据总线宽度为 64 位。这可以在一个周期内写入 8 位、16 位、32 位、64 位数据。

备注

ECC 特性：C29x CPU 支持 16/32/64 位的 ECC 粒度，并具有单位误差校正功能。双位误差检测导致 CPU 进入故障状态。当校正单位误差时，CPU 会使流水线停止 1 个周期。

调试数据读取总线：C29x CPU 具有类似于数据读取总线的专用调试数据读取总线。调试数据读取总线的数据总线宽度为 64 位。该总线在单周期内可以读取 8 位、16 位、32 位和 64 位数据。功能安全和信息安全单元 (SSU) 根据安全设置允许或阻止调试访问。

调试数据写入总线：C29x CPU 具有类似于数据写入总线的专用调试数据写入总线。调试数据写入总线的数据总线宽度为 64 位。该总线在单周期内可以写入 8 位、16 位、32 位和 64 位数据。功能安全和信息安全单元 (SSU) 根据安全设置允许或阻止调试访问。

中断总线：C29x CPU 中断总线可处理复位、NMI、RTINT、INT 中断信号和中断向量。

ERAD 接口总线：使用 ERAD（实时分析和诊断）模块在 C29x CPU 的外部实现断点和观察点。该总线用于将 ERAD 与 C29x CPU 连接。

SSU 接口总线：安全实现通过 SSU 接口总线与 C29x CPU 紧密耦合。

错误接口总线：使用错误接口总线将程序读取错误、数据读取错误和数据写入错误连接到错误聚合器/ESM。

1.4 存储器映射

C29x CPU 具有一个专用栈指针 (SP = A15)，可访问 CPU 的完整 32 位地址范围。

C29x CPU 支持单独的程序、数据读取和数据写入总线。存储器与一个 4GB 映像统一。在 C29x CPU 上，每个外设实例都映射在 4KB 地址范围内。

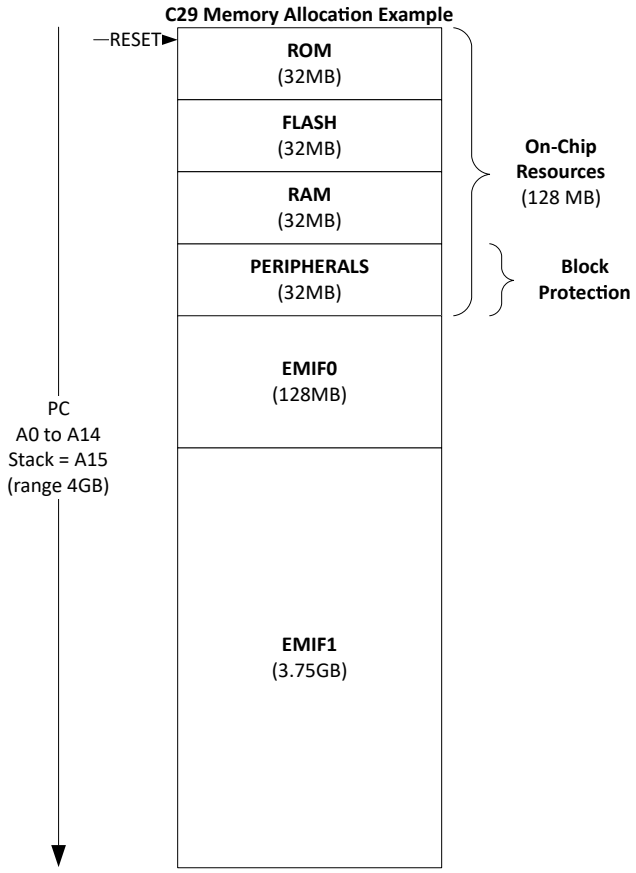

图 1-2. 存储器映射

块保护：此特性可保护外设寄存器的读取和写入操作顺序并避免流水线影响。

章节 2

中央处理单元 (CPU)

中央处理单元 (CPU) 负责控制程序流和指令处理。CPU 执行算术、布尔逻辑、乘法和移位运算。执行有符号的数学运算时，CPU 使用二进制补码表示法。本章对 CPU 的架构、寄存器和主要功能进行了说明。

2.1 C29x CPU 架构

C29x CPU 采用 VLIW（超长指令字）架构，并配备全面保护式流水线。CPU 支持多种指令大小（16/32/48 位）。CPU 还支持可变指令包大小，每个指令包可包含多达 8 条并行执行的指令。例如，CPU 架构能够并行执行多达 8 条 16 位指令。这由 CPU 内可以同时执行的多个功能单元实现。工作寄存器总共有 64 个，分为三个不同类别（Ax、Dx 和 Mx 寄存器组），用于支持 CPU 中的并行操作。除了工作寄存器外，CPU 还包含多个状态寄存器（DSTS、EST 和 ISTS），用于维护与执行相关的信息和与中断上下文相关的信息。

2.1.1 特性

以下列出了 C29x CPU 的主要特性：

- **简便易用：**
  - 字节可寻址 CPU。
  - 具有 4GB 地址范围的线性和统一存储器映射。
  - 全面保护式流水线：9 级流水线，可防止对同一位置进行无序写入和读取。
  - 在无缓存存储器的情况下实现确定性执行和出色性能。
- **改进并行性：**
  - 并行执行 1 到 8 条指令。
  - 并行执行定点、浮点和寻址运算。
  - 多个并行功能单元。
  - 专门的运算，可更大限度地减少不连续性并加速决策代码（例如 if-then-else 语句和 switch 语句）。
  - 面向实时控制的专业运算（例如，三角运算和多相矢量转换运算）。
- **提高总线吞吐量：**
  - 每个周期能够获取多达 128 位指令包。
  - 每个周期能够执行 8/16/32/64 位双读取操作和单写入操作。
  - 改进的寻址模式减少了内存和外设资源访问的开销。
  - 改进的流水线使 CPU 能够访问更多的 0 等待存储器，从而实现超高性能。
- **代码效率：**
  - 支持可变长度指令集（16 位、32 位和 48 位指令）。
  - 丰富的指令集通过超简洁的指令优化了常见的运算。
- **硬件中实现代码隔离的 ASIL-D 级别安全功能：**
  - 锁步内核能够在分离锁定模式下独立执行（用作单独内核）或进行锁步执行（用于提供冗余）。
  - 集成 ECC 逻辑
  - 在硬件中集成内存管理 (MPU) 和保护机制，从而更大限度地提高 MIPS。
  - 独立的代码线程实现了完全隔离与保护（包括软件栈）。
- **硬件中的多区域安全：**
  - 运行时内容保护和代码的 IP 保护。
  - 为每个区域设置单独的密码以控制访问。
- **增强调试和跟踪功能：**
  - 专用数据记录和代码流跟踪指令。
  - 跟踪数据能够记录在片上 RAM 中或通过串行通信外设导出。

2.1.2 方框图

图 2-1 展示了 C29x CPU 的方框图。

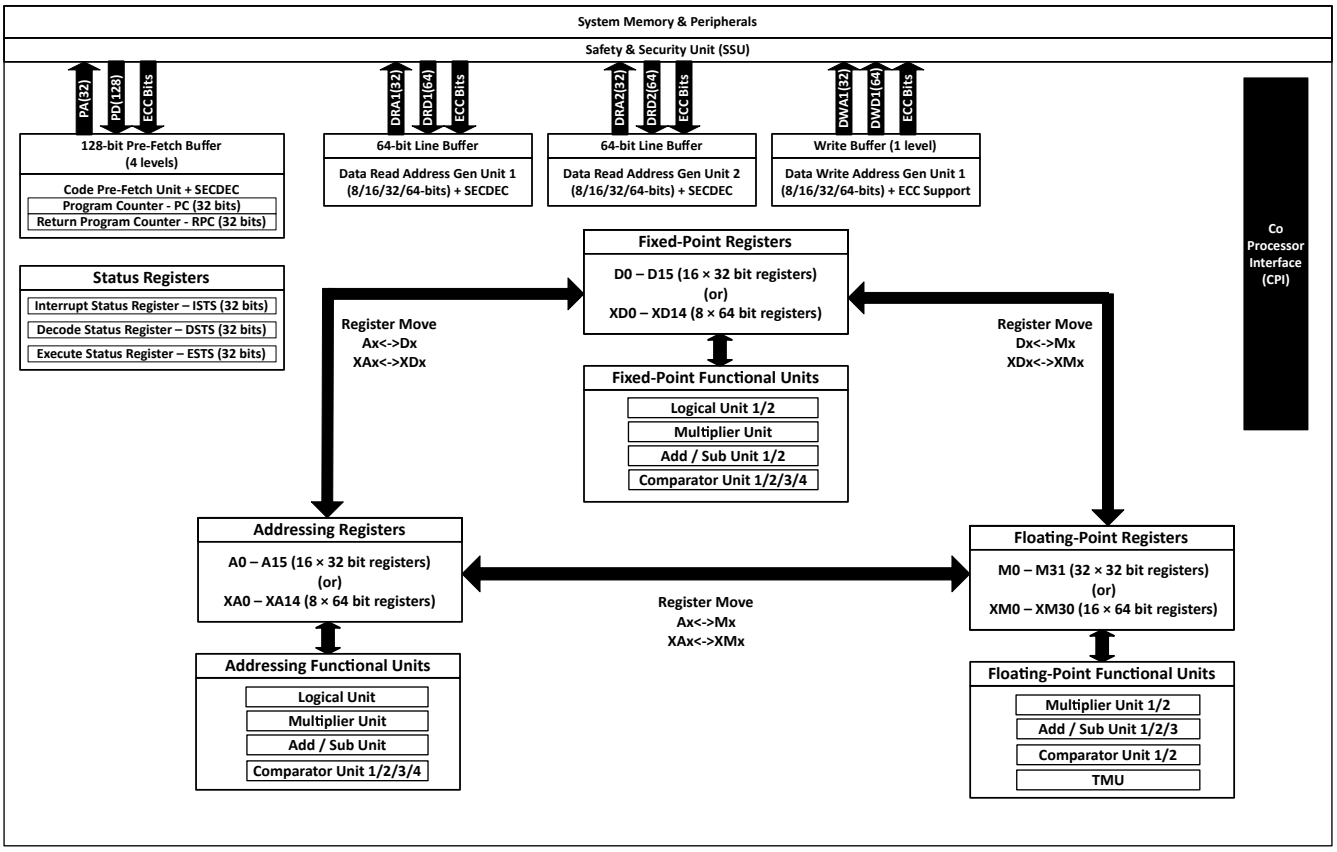

图 2-1. C29x CPU 方框图

2.2 CPU 寄存器

C29x CPU 内核由以下 CPU 寄存器组成：

- 寻址寄存器 (Ax/XAx)
  - 栈指针 (A15 = SP) 寄存器
- 定点寄存器 (Dx/XDx)
- 浮点寄存器 (Mx/XMx)
- 程序计数器 (PC)
- 返回程序计数器 (RPC)
- 状态寄存器
  - 中断状态寄存器 (ISTS)
  - 解码状态寄存器 (DSTS)
  - 执行状态寄存器 (ESTS)

2.2.1 寻址寄存器 (Ax/XAx)

表 2-1. 寻址寄存器 (Ax/XAx)

寄存器		尺寸	说明
A0	XA0	32 位	**寻址寄存器（16 个 Ax、8 个 XAx）：** • Ax 寄存器主要用于寻址操作。所有寻址模式都在 Ax 寄存器上运行。 • 有 16 个 32 位寻址寄存器 (A0-A15) 或 8 个 64 位寻址寄存器对 (XA0-XA14)。 • Ax 寄存器也可用于执行 MPY、ADD、COMP、SHIFT 和 AND/OR/XOR 运算。这些寄存器用于为 C29x CPU 存储器空间 (4GB) 生成 32 位地址。 • 这些寄存器可以用作单独的 32 位寄存器，也可用于 64 位寄存器对，作为与存储器之间的 64 位加载/存储操作，或 64 位寄存器到寄存器移动（8 对，XA0、XA2 到 XA14）。 • 寄存器 A15 专用于栈指针 (A15 = SP) 寄存器。 **复位后的值：0x0000 0000**
A1		32 位	
A2	XA2	32 位	
A3		32 位	
A4	XA4	32 位	
A5		32 位	
A6	XA6	32 位	
A7		32 位	
A8	XA8	32 位	
A9		32 位	
A10	XA10	32 位	
A11		32 位	
A12	XA12	32 位	
A13		32 位	
A14	XA14	32 位	
A15 (SP)		32 位	

2.2.2 定点寄存器 (Dx/XDx)

表 2-2. 定点寄存器 (Dx/XDx)

寄存器		尺寸	说明
D0	XD0	32 位	**定点寄存器（16 个 Dx，8 个 XDx）：** • Dx 寄存器主要用于执行定点数据运算。 • 共有 16 个 32 位定点寄存器 (D0-D15) 或 8 个 64 位定点寄存器对 (XD0-XD14)。 • 这些寄存器可以用作单独的 32 位寄存器 (Dx)，也可以用于 64 位寄存器对 (XDx) 进行 64 位运算，执行与存储器之间的 64 位加载/存储操作，或执行 64 位寄存器到寄存器移动（8 对、XD0、XD2 到 X14）。 **复位后的值：0x0000 0000**
D1		32 位	
D2	XD2	32 位	
D3		32 位	
D4	XD4	32 位	
D5		32 位	
D6	XD6	32 位	
D7		32 位	
D8	XD8	32 位	
D9		32 位	
D10	XD10	32 位	
D11		32 位	
D12	XD12	32 位	
D13		32 位	
D14	XD14	32 位	
D15		32 位	

2.2.3 浮点寄存器 (Mx/XMx)

表 2-3. 浮点寄存器 (Mx/XMx)

寄存器		尺寸	说明
M0	XM0	32 位	**浮点寄存器（32 个 Mx、16 个 XMx）：** • Mx 寄存器主要用于执行浮点数据运算。 • 有 32 个 32 位寻址寄存器 (M0-M15) 或 16 个 64 位寻址寄存器对 (XM0-XM14)。 • 这些寄存器可以用作单独的 32 位寄存器 (Mx)，也可以用于 64 位寄存器对 (XMx) 进行 64 位运算，执行与存储器之间的 64 位加载/存储操作，或执行 64 位寄存器到寄存器移动（32 对、XM0、XM2 到 XM30）。 **复位后的值：0x0000 0000**
M1		32 位	
M2	XM2	32 位	
M3		32 位	
M4	XM4	32 位	
M5		32 位	
M6	XM6	32 位	
M7		32 位	
M8	XM8	32 位	
M9		32 位	
M10	XM10	32 位	
M11		32 位	
M12	XM12	32 位	
M13		32 位	
M14	XM14	32 位	
M15		32 位	
M16	XM16	32 位	
M17		32 位	
M18	XM18	32 位	
M19		32 位	
M20	XM20	32 位	
M21		32 位	
M22	XM22	32 位	
M23		32 位	
M24	XM24	32 位	
M25		32 位	
M26	XM26	32 位	
M27		32 位	
M28	XM28	32 位	
M29		32 位	
M30	XM30	32 位	
M31		32 位	

2.2.4 程序计数器 (PC)

当流水线已满时，32 位 PC 始终指向当前正在处理的指令。PC 从低存储器到高存储器递增，并且一直指向下一个可执行指令包。

2.2.5 返回程序计数器 (RPC)

当执行调用或处理低优先级中断 (INT) 时，该寄存器保存返回地址。之前的 RPC 值保存在栈上。在执行 RET（从函数返回）或 RETI.INT（从低优先级中断返回）操作时，从 RPC 寄存器中获取返回地址，然后使用保存在栈上的值（作为 CALL 或 INT 操作）恢复 RPC。

2.2.6 状态寄存器

C29x CPU 内核支持三个状态寄存器：包含标志位和控制位的 ISTS (节 2.2.6.1)、DSTS (节 2.2.6.2) 和 ESTS (节 2.2.6.3)。EST 和 DSTS 状态寄存器会存储到数据存储器中并从数据存储器中加载，从而为子例程保存和恢复 CPU 的状态。

2.2.6.1 中断状态寄存器 (ISTS)

表 2-4. 中断状态寄存器 (ISTS)

位	位域	复位值	说明
0	INTF	0h	当 PIPE 生成 INT 中断时，将设置此标志
1	RTINTF	0h	当 PIPE 生成 RTINT 中断时，将设置此标志
2	NMIF	0h	当 PIPE 生成 NMI 中断时，将设置此标志
3-7	RESERVED	0h	RESERVED
8-15	ATOMIC 计数器	0h	**ATOMIC 计数器：**执行 ATOMIC #N 操作时，计数器将载入指定的计数值 #N 然后，计数器在每次指令包执行时开始递减。中断会被阻止，直到计数器达到零。支持的最大 #N 值为 64 个指令包。
16-19	CURRSP	0h	**当前栈指针：**C29x CPU 系统支持多个软件栈。该字段反映当前的活动栈。
20-23	INTSP	0h	**中断栈指针：**INT 中断只能从一个所选栈（INTSP 字段所指示的栈）中执行。该值在中断控制器 (PIPE) 中编程。复位值由 PIPE 驱动的值确定。
24-26	RESERVED	0h	RESERVED
27-30	CURRLINK	0h	**当前链接：**C29x CPU 信息安全和功能安全系统支持“链接”的概念（如章节 5 所述）。该字段反映 D2 阶段的当前活动链接值。
31	RESERVED	0h	RESERVED

2.2.6.2 解码阶段状态寄存器 (DSTS)

表 2-5. 解码阶段状态寄存器 (DSTS)

位	位域	复位值	说明
0	A.Z	0h	**Ax 寄存器操作标志：**这些标志是在涉及 Ax 寄存器的定点运算上设置的。经过测试的条件： A.EQ：等于零 A.NEQ：不等于零 A.GT：大于零 A.GEQ：大于或等于零 A.LT：小于零 A.LEQ：小于（或）等于零 A.HI：更高 A.HIS：更高（或）相同 A.LO：更低 A.LOS：更低或相同 A.EQANDNZ：等于和非零（适用于字符串搜索） A.NEQORZ：不等于或零（适用于字符串搜索）
1	A.N	0h	
2	A.C	0h	
3	A.ZV	0h	
4-5	RESERVED	0h	RESERVED
6	DBGM	0h	调试掩码位，启用或禁用调试请求。
7-10	CLINK(1)	0h	用于指示受保护 CALL 操作的来源。
11	RESERVED	0h	RESERVED
12	TA0	0h	**Ax 寄存器测试标志：**这些测试标志可以通过测试 Ax 操作标志来存储多个条件。然后，这些测试标志可用于组合测试条件的多种组合。这可以减少多个条件分支操作。经过测试的条件： TAx.Z TAx 等于零 TAx.NZ TAx 不等于零 TA.MAP(#x16ta) 使用 4:1 LUT 组合测试 TAx 标志
13	TA1	0h	
14	TA2	0h	
15	TA3	0h	
16	INTE	0h	中断 (INT) 使能位
17-18	INTS(2)	0h	**中断状态：**这些位表示当前有效的中断 ISTS = 0 : 主代码有效 ISTS = 1 : INT 有效 ISTS = 2 : RTINT 有效 ISTS = 3 : NMI 有效
19-26	ISR PRIORITY(2)	FFh	ISR PRIORITY 级别介于 0（最高优先级）到 255（最低优先级）之间。
27-30	RLINK (1)	0h	用于指示受保护 RET 操作的来源。
31	RESERVED	0h	RESERVED

(1) CLINK 和 RLINK 仅由硬件更新。加载、移动和屏蔽操作不会更改这些字段的状态。
(2) INT 和 ISR 优先级仅由硬件和 RETI.INT 指令更新。任何其他加载、移动和屏蔽操作不会更改这些字段的状态。

2.2.6.3 执行阶段状态寄存器 (ESTS)

表 2-6. 执行阶段状态寄存器 (ESTS)

位	位域	复位值	说明
0	D.Z	0h	**Dx 寄存器运算标志：**这些标志是在涉及 Dx 寄存器的定点运算上设置的。经过测试的条件： D.EQ 等于零 D.NEQ 不等于零 D.GT 大于零 D.GEQ 大于或等于零 D.LT 小于零 D.LEQ 小于或等于零 D.HI 更高 D.HIS 更高或相同 D.LO 更低 D.LOS 更低或相同 D.EQANDNZ 等于和非零 (适用于字符串搜索) D.NEQORZ 不等于或零 (适用于字符串搜索) D.OV 整数溢出 D.OVNEG 整数溢出为负
1	D.N	0h	
2	D.C	0h	
3	D.ZV	0h	
4	D.OV(1) (2) (3)	0h	
5	D.OVNEG(1) (2) (3)	0h	
6-7	RESERVED	0h	
8	M.ZF	0h	
9	M.NF		
10	M.LUF(1) (2)	0h	
11	M.LVF(1) (2)		
12	TDM0	0h	
13	TDM1		
14	TDM2	0h	
15	TDM3		
16	RNDF32	0h	
17	RNDF64		
18	IDIV.Z	0h	
19	IDIV.N		
20	IDIV.TF	0h	
21	FDIV.TF		
22	FDIV.N	0h	
23	TMU.TF		
24-31	RESERVED	0h	RESERVED

(1) 在 C29x CPU 上，所有标志 (除了 D.OV、D.OVNEG、M.LUF 和 M.LVF) 只受比较 (CMP)、测试位 (TBIT) 或测试标志 (TFLG) 类型运算的影响。加载/存储、MPY、ADD、SUB、SHIFT、AND、OR 和 XOR 类型运算不会影响标志。

(2) D.OV、D.OVNEG、M.LUF 和 M.LVF 是粘滞标志，也就是说，一旦被设定，这些标志在被软件清零前保持被置位。

(3) D.OV 和 D.OVNEG 被设定为一个对。D.OVNEG 在第一次出现 D.OV 时更新，也就是说，如果指令序列多次更新 D.OV，D.OVNEG 会在第一次出现 D.OV 时捕获溢出状态。

2.3 指令打包

C29x CPU 具有可变大小指令集。支持的指令大小为 16 位、32 位和 48 位。CPU 的 VLIW 架构允许在单周期内发出多条指令。并行执行的指令数量在构建时确定，并且所有并行指令都打包在单个指令包中。本节介绍了指令包的组成和结构。允许的最大指令包大小为 128 位。因此，只要不超过最大指令包大小，16 位、32 位和 48 位指令的任何组合均可以组成指令包。

以下是指令包内有效指令组合示例的非详尽列表：

- 8 * 16 位指令。
- 4 * 32 位指令。
- 2 * 32 位、1 * 48 位、1 * 16 位指令。
- 3 * 32 位、2 * 16 位指令。

表 2-7 展示了三个可能指令大小的结构。

表 2-7. 指令大小和编码

指令大小	字 0（低地址）				字 1（下一个地址）	字 2（下一个地址）
	15	14	13	12:0	31:16	47:32
16	I_Link	1	操作码			
32	I_Link	0	1	操作码	16 位参数	
48	I_Link	0	0	操作码	32 位参数的低 16 位	32 位参数的高 16 位

2.3.1 独立指令和限制

以下是限制条件：

- 不连续指令不能包含在延迟时隙中。无硬件检查。但是，汇编器应标记此错误。
- IDLE 指令不能在延迟时隙中执行。
- IDLE 指令不能放置在 XC 覆盖的指令包中
- IDLE 指令包不能被并行化。
- PRESERVE 指令只能与受保护的调用或受保护的分支或受保护的返回并行执行。无硬件检查。但是，汇编器应标记此错误。
- EMUSTOP0 不能包含在延迟时隙中。
- 延迟时隙中不允许包含多个指令包的 XC 包。
- ECCSELFTEST 不能仅以独立方式并行和执行。
- CALL 指令的延迟时隙中不得有 MOV Ax、RPC 和 LD.32 RPC、@MEM。延迟时隙 3 中返回地址的 RPC 负载不受保护。
- XC/XCP 指令无法与 ISRn.PROT/ENTRYn.PROT/EXITn.PROT 并行执行

2.3.2 指令超时

由于打包错误、不可纠正的错误或错误的 #delay 设置，指令解码可能无法形成合法的指令包。在这种情况下，CPU 进入故障状态时会使用超时逻辑。只要新指令进入流水线（或）进入 HALT，超时计数器就会复位。当 D2 中没有指令时，超时计数器递增。如果超时计数器超过指定的超时值，则 CPU 将进入故障状态。

2.4 栈

C29x CPU 包含以下栈：

1. 软件栈
2. 受保护的调用栈
3. 实时中断栈

2.4.1 软件栈

可用寻址寄存器中的寄存器 A15 专用于栈指针寄存器 (SP = A15)。A15 寄存器可以访问 CPU 的完整 32 位地址范围 (4GB) (请参阅图 2-2) 。

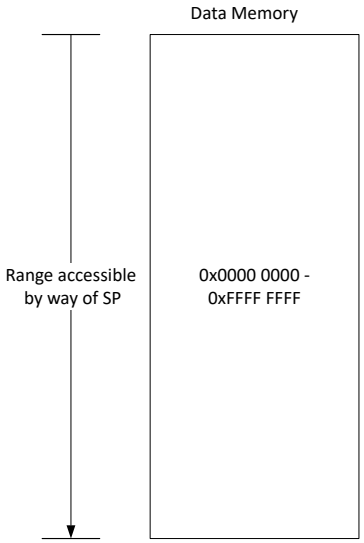

图 2-2. 栈指针的地址范围

栈的操作如下：

- 栈从低位存储器地址增长到高位存储器地址。
- 栈指针始终指向栈中的下一个空位置。
- 复位时，栈指针初始化为 0x0000 0000。
- C29x 栈指针始终与 64 位字边界对齐。

备注

C29x 栈指针 (SP = A15) 必须始终与 64 位字边界对齐。任何未对齐的栈都会导致 CPU 进入故障状态。

如果需要在软件栈上传递参数，则汇编代码结构如下所示：

```
; Allocate Parameter Space On Stack:
ADD.U16 A15, A15, #PARAMETER_SPACE

;....pass parameters on stack…

;....pass parameters in registers…

CALL.PROT @func
; 32-bit RPC automacally pushed on stack
; A15 = A15 + 8 (stack pointer incremented by 8 bytes = 64-bits)
; There is a 32-bit hole in the stack that can be used to save
```

```
; De-allocate Parameter Space From Stack
SUB.U16 A15, A15, #PARAMETER_SPACE
```

如果需要在软件栈上分配局部变量，则代码结构如下所示：

```
func:
    ; Allocate Local Variable Space On Stack:
    ADD.U16 A15, A15, #PARAMETER_SPACE

    ; ... save on stack any registers used that need to be preserved across call..

    ; ... function code....

    ; ... restore from stack any registers used that need to be preserved across call..

    ; De-allocate Local Variable Space From Stack
    SUB.U16 A15, A15, #PARAMETER_SPACE


    RET.PROT
    ; 32-bit RPC automacally popped from stack
    ; A15 = A15 - 8 (stack pointer decremented by 8 bytes = 64 bits)
```

在进行正常函数调用时：返回程序计数器（RPC，保持先前的返回地址）寄存器内容被压入栈指针 (SP = A15) 指向的软件栈上。然后，使用新的返回地址初始化 RPC 寄存器。

在正常函数返回时：从 RPC 寄存器读取返回地址。然后，使用软件栈上的值恢复 RPC 寄存器。

有关 RPC 的更多详细信息，请参阅节 2.2

2.4.2 受保护的调用栈

受保护调用栈是一种专用硬件栈，用于进行受保护的函数调用和返回。这个栈直接由 CPU 控制并且用户代码无法访问。C29x CPU 的基本保护概念基于链接、栈和区域。受保护的函数调用和返回是通过当前执行代码向位于不同栈中的另一个函数进行函数调用的方法。C29x 安全架构允许使用指令 ENTRY1.PROT 和 ENTRY2.PROT 来定义合法可调用函数标签。这可确保另一个栈中的代码只能调用或跳转到包含指令包“ENTRY1.PROT || ENTRY2.PROT”的标签。这可以防止恶意代码在没有许可的情况下随机进入代码区域。允许的受保护调用嵌套最多可达到受保护调用栈支持的级别数。表 2-8 展示了跨栈的代码执行规则。

受保护的调用栈指针 (PSP) 寄存器：PSP 寄存器跟踪受保护调用栈的使用情况并显示受保护调用栈指针的当前值。在进行受保护调用 (CALL.PROT) 和受保护返回 (RET.PROT) 时，此寄存器会由硬件自动递增和递减。

受保护调用栈指针 (WARNPSP) 寄存器的警告级别：此 WARNPSP 是一个用户可配置的寄存器，可以针对受保护的栈溢出检测发出预警。当 PSP 寄存器 >= WARNPSP 寄存器时，向 ESM 生成错误信号。

受保护的最大调用栈指针 (MAXPSP) 寄存器：MAXPSP 寄存器不是用户可配置的寄存器。当 PSP 寄存器 = MAXPSP 寄存器时，CPU 在受保护的调用栈已满时进入故障状态。

表 2-8. 跨栈的代码执行规则

程序流操作	注释和 CPU 操作
线性代码在同一链接中执行	任何限制条件下允许
在同一链接中执行分支、调用和返回	
跨不同链接的分支、调用和返回，但在同一栈内执行	
受保护函数返回 (RET.PROT)，其中返回地址位于不同于当前栈的栈上	
源和目标位于同一栈上的受保护函数调用 (CALL.PROT @label/Ax)	
受保护函数返回 (RET.PROT)，其中返回地址位于同一栈上	
线性代码执行跨链接但位于同一栈内	不允许，CPU 进入故障状态。
源和目标位于不同栈上的分支	
源和目标位于不同栈上的函数调用 (CALL{D} @label/Ax)	
执行函数返回指令 (RET{D} /RET{D} <addr1>)，其中返回地址位于不同于当前栈的栈上	
实时中断 (RTINT) 和 NMI	这在硬件中进行处理，无需在用户代码中考虑任何因素。中断服务例程可以驻留在同一个或不同的链接/栈/区域中。
中断 (INT)	ISR 必须位于同一栈上。否则，CPU 进入故障状态。

2.4.3 实时中断/NMI 栈

实时中断栈 (RTINT) 是实时中断 (RTINT) 和不可屏蔽中断 (NMI) 使用的专用硬件栈。有关各种中断类型之间差异的详细信息，请参阅章节 3。当这些中断中的任一个被触发时，所有 C29x CPU 工作寄存器（Ax、Dx、Mx、RPC、DSTS 和 EST）和返回地址都将在 8 个周期内保存在 RTINT 栈上，并在执行 RETI.RTINT 指令时在 8 个周期内恢复。允许 RTINT 嵌套的级别数不超过实时中断栈所支持的级别数减 1 级，NMI 中断始终具有一个保留级别。

实时中断栈指针 (RTISP) 寄存器：RTISP 寄存器跟踪栈使用情况并显示实时中断栈指针的当前值。当实时中断或 NMI 中断被触发时，此寄存器由硬件自动递增，当执行 RETI.RTINT 指令时，此寄存器递减。

实时中断栈指针 (WARNISP) 寄存器的警告级别：此 WARNISP 是一个用户可配置寄存器，当 RTISP 寄存器大于或等于 WARNISP 寄存器值时，可以针对实时中断栈溢出检测发出预警。

最大实时中断栈指针 (MAXISP) 寄存器：MAXISP 寄存器不是用户可配置的寄存器。当 ISP 寄存器等于 MAXISP 寄存器时，CPU 会在实时中断栈已满时进入故障状态。

有关与实时中断栈相关的寄存器的更多详细信息，请参阅节 3.4.3。

备注

实时中断栈指针 (ISP) 寄存器、实时中断栈指针 (WARNISP) 寄存器的警告级别和最大实时中断栈指针 (MAXISP) 寄存器都是 PIPE（外设中断优先级和扩展）中的寄存器。

章节 3
中断

TEXAS INSTRUMENTS

什么是中断？

中断是硬件或软件驱动的信号，可导致 CPU 暂停当前的程序序列并执行子例程。通常，中断由需要向 C29x CPU（例如 ADC、DAC、ePWM 和其他处理器）提供数据或从其获取数据的外设或硬件器件生成。中断还可以指示已发生特定事件（例如，计时器已完成计数）。

C29x CPU 中断

在 C29x CPU 上，系统中有四种类型的中断线路。下面按照从最高优先级到最低优先级的顺序列出了这些中断线路：

- RESET
- NMI（不可屏蔽中断）
- RTINT（实时中断，可屏蔽）
- INT（低优先级中断、可屏蔽、可禁用）

这四条中断线会处理器件上的所有中断和异常。所有 C29x 器件都附带一个 PIPE（外设中断优先级和扩展）模块，可为 RTINT 和 INT 线路提供额外的优先级划分和仲裁。有关 PIPE 模块的详细信息，请参阅 *F29H85x 和 F29P58x 实时微控制器技术参考手册*。

3.1 CPU 中断架构方框图

图 3-1 展示了 C29x CPU 中断架构的方框图。

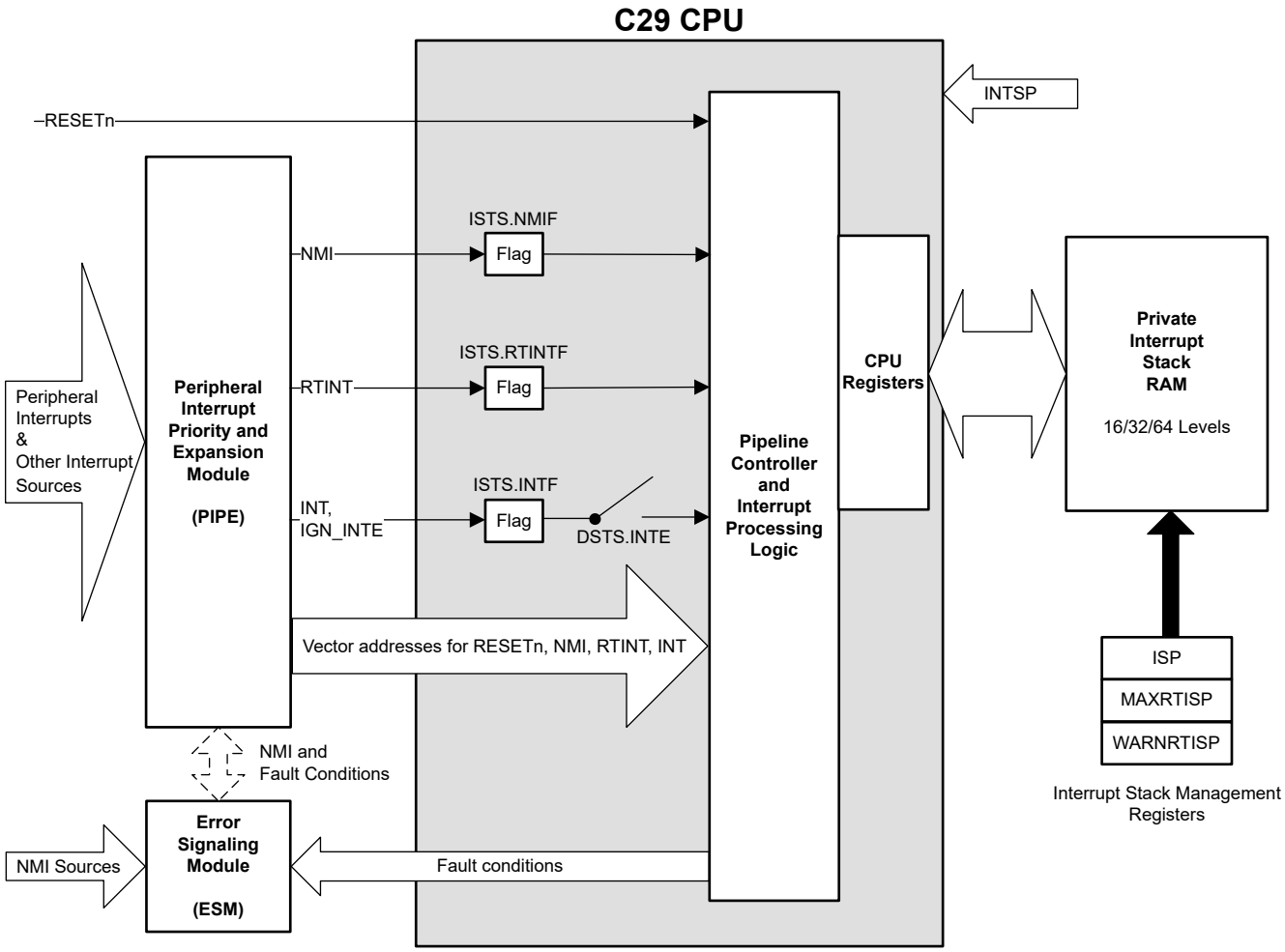

图 3-1. C29x CPU 中断架构方框图

3.2 RESET、NMI、RTINT 和 INT

本节介绍了 C29x CPU 架构中可用的四条中断线路。

3.2.1 RESET（CPU 复位）

CPU 复位是最高优先级的中断线路，当 RESETn 线路接收到低电平有效信号时，会发生 CPU 复位。这会导致 CPU 在内部进行硬件复位。无法中止或嵌套。

流水线中的所有当前和待处理操作都将中止，并且在复位期间清除该流水线。

如表 3-1 所示，所有 CPU 寄存器都被重置为复位值（全 0）。

表 3-1. CPU 寄存器复位值

寄存器	复位值
A0 至 **A15**	0x0000 0000
D0 至 **D15**	0x0000 0000
M0 至 **M31**	0x0000 0000
DSTS	0x07F8 0000
ESTS	0x0000 0000
RPC	0x0000 0000
ISTS	0x0000 0000

3.2.1.1 所需指令 (RESET)

NMI 和 RTINT 中断可能会使各自的中断服务例程 (ISR) 驻留在不同的链接/栈中。因此，NMI 和 RTINT ISR 要求每个向量地址的第一个指令数据包包含 (ISR1.PROT || ISR2.PROT) 指令。CPU 流水线控制硬件会检查这些所需的指令，如果这些指令不是 ISR 的第一个指令包，则生成故障。这些所需指令由编译器自动插入，但必须配置为在单独的安全设置文件中针对相应的向量执行此操作。有关更多详细信息，请参阅节 3.6。

ISR1.PROT 还会通过执行以下操作来初始化指向适当栈的栈指针 (A15)：A15 = SECSPn（其中 n 是 ISTS.CURRSP 指示的当前栈）。

有关链接/栈/区域的安全影响以及 CPU 中断的存储器空间的详细信息，请参阅节 3.6。

3.2.2 NMI（不可屏蔽中断）

NMI（不可屏蔽中断）是第二高优先级中断线路，并且接收系统异常中断。

该 NMI 输入线路用于任何器件级关键条件以及 CPU 内部或外部需要立即处理的各种故障。

3.2.2.1 阻止和屏蔽 (NMI)

CPU 中无法屏蔽或阻止 NMI。在 CPU 内没有针对 NMI 线路的全局启用/禁用位。因此，在 NMI 线路上接收到的任何中断都会直接传递给 CPU 进行优先级排序。然后在中断类型（NMI、RTINT 和 INT 线路）中确定优先级。NMI 始终具有最高优先级，并在当前执行的任何 RTINT 或 INT 内有效。RTINT 或 INT ISR 中的 ATOMIC 指令无法阻止或禁止 NMI 有效。ATOMIC 指令对 NMI 没有影响。

3.2.2.2 信号传播 (NMI)

CPU 的 NMIn 输入通常由位于 CPU 外部的错误信令模块 (ESM) 单元生成。此类单元可实现系统故障的聚合和优先级排序。即使是 CPU 内部发生的故障情况，也会将故障信息传播到通用器件电平聚合器单元，然后生成 NMI 返回到 CPU。

NMIn 输入在 CPU 内锁存，并且处理时的优先级高于所有其他中断类型（复位事件除外）。

3.2.2.3 栈 (NMI)

此中断线路使用受保护的实时中断栈进行背景保存和恢复。该 SSU 保护（功能安全和信息安全单元）栈具有保护功能，可在 PIPE 模块请求嵌套时防止栈溢出。WARNRTISP 和 MAXRTISP CPU 寄存器可在 C29x CPU 系统中用于此目的。

这种保护将 RTINT 的嵌套限制在 RTINT 栈所支持的级别数减去一个级别（始终为 NMI 中断而保留）。

为安全起见，RTINT 栈的 SSU 保护设计为不可见栈内容。寄存器也被置为零，这样就不能监测中断服务之前所发生的情况。

有关使用 WARNRTISP 和 MAXRTISP 寄存器进行栈溢出保护的详细信息，请参阅节 2.4。

3.2.2.4 所需指令(NMI)

NMI 和 RTINT 中断可能会使各自的中断服务例程 (ISR) 驻留在不同的链接/栈中。因此，NMI 和 RTINT ISR 要求每个向量地址的第一个指令数据包包含 (ISR1.PROT || ISR2.PROT) 指令。CPU 流水线控制硬件会检查这些所需的指令，如果这些指令不是 ISR 的第一个指令包，则生成故障。这些所需指令由编译器自动插入，但必须配置为在单独的安全设置文件中针对相应的向量执行此操作。有关更多详细信息，请参阅节 3.6。

ISR1.PROT 还会通过执行以下操作来初始化指向适当栈的栈指针 (A15)：A15 = SECSPn（其中 n 是 ISTS.CURRSP 指示的当前栈）。

有关链接/栈/区域的安全影响以及 CPU 中断的存储器空间的详细信息，请参阅节 3.6。

3.2.3 RTINT（实时中断）

RTINT（实时中断）是第三高优先级中断线路，接收由中断扩展和聚合单元（对于大多数 C29x CPU 系统，为 PIPE 模块）驱动的信号。

3.2.3.1 阻止和屏蔽 (RTINT)

RTINT 源可以被屏蔽，但连接到 CPU 的实际 RTINT 线路永远不会被用户代码阻止/禁用。在 CPU 内没有针对 RTINT 线路的全局启用/禁用位。因此，在 RTINT 线路上接收到的任何中断都会直接传递给 CPU 进行优先级排序。然后，确定 NMI 或 INT 线路上任何中断的优先级。RTINT 信号线只能通过使用 INT ISR 中的 ATOMIC 指令来停止嵌套在 INT 内，并且该指令只能用于有限数量的指令包。但是，可以在使用外部 PIPE 模块使中断到达 RTINT 线路之前优先处理/阻止中断 ISR。

3.2.3.2 信号传播 (RTINT)

PIPE 模块为 RTINT 和 INT 线路提供外部中断聚合和仲裁。这允许许多信号被归类为实时中断 (RTINT) 或低优先级中断 (INT)，然后在传递到 CPU 的 RTINT 或 INT 中断线路之前进行优先级排序。

PIPE 有效地多路复用了单个 RTINT CPU 中断线路，以便能够按适当的顺序从多个传入的 RTINT 中断中接收。

该模块允许在信号到达 CPU 的 RTINT 线路之前启用和禁用 RTINT 信号。该模块还允许嵌套功能以及归为 RTINT 的其他中断。有关 PIPE 模块功能的详细信息，请参阅 *F29H85x 和 F29P58x 实时微控制器技术参考手册*。

3.2.3.3 栈 (RTINT)

此中断线路使用受保护的实时中断栈进行背景保存和恢复。该 SSU 保护（功能安全和信息安全单元）栈具有保护功能，可在 PIPE 模块请求嵌套时防止栈溢出。WARNRTISP 和 MAXRTISP CPU 寄存器可在 C29x CPU 系统中用于此目的。

这种保护将 RTINT 的嵌套限制在 RTINT 栈所支持的级别数减去一个级别（始终为 NMI 中断而保留）。

为安全起见，RTINT 栈的 SSU 保护设计为不可见栈内容。寄存器也被置为零，这样就不能监测中断服务之前所发生的情况。

有关使用 WARNRTISP 和 MAXRTISP 寄存器进行栈溢出保护的详细信息，请参阅节 2.4。

3.2.3.4 所需指令 (RTINT)

NMI 和 RTINT 中断可能会使各自的中断服务例程 (ISR) 驻留在不同的链接/栈中。因此，NMI 和 RTINT ISR 要求每个向量地址的第一个指令数据包包含 (ISR1.PROT || ISR2.PROT) 指令。CPU 流水线控制硬件会检查这些所需的指令，如果这些指令不是 ISR 的第一个指令包，则生成故障。这些所需指令由编译器自动插入，但必须配置为在单独的安全设置文件中针对相应的向量执行此操作。有关更多详细信息，请参阅节 3.6。

ISR1.PROT 还会通过执行以下操作来初始化指向适当栈的栈指针 (A15)：A15 = SECSPn（其中 n 是 ISTS.CURRSP 指示的当前栈）。

有关链接/栈/区域的安全影响以及 CPU 中断的存储器空间的详细信息，请参阅节 3.6。

3.2.4 INT（低优先级中断）

INT（低优先级中断）是最低优先级中断线路，接收由中断扩展和聚合单元（对于大多数 C29x CPU 系统，为 PIPE 模块）驱动的信号。该中断线路通常用于低优先级操作和任务调度器。

3.2.4.1 阻止和屏蔽 (INT)

INT 源可以被屏蔽，INT 线路也可以由用户代码使用 DSTS.INTE 使能位被阻止/禁用。如果启用了 DSTS.INTE，则在 INT 线路上接收到的任何中断都会直接传递给 CPU 以进行优先级设置。然后，确定 NMI 或 RTINT 线路上的中断的优先级。为了防止 RTINT 中断嵌套在 INT 中断中，ATOMIC 指令可用于有限数量的指令包。

进入 INT ISR 后，会使用 DSTS.INTE 位自动禁用进一步的 INT。为了实现嵌套，使用 ENINT 指令启用中断。还有一条 DISINT 指令用于再次禁用 INT 线路。

C29x CPU 还提供一个称为监控器中断的特殊 INT。监控器中断本质上是一个 INT，可以覆盖 DSTS.INTE 设置。例如，监控器中断可以是某个任务监控器中断，它要求中断不会被 DSTS.INTE 的错误设置阻止。

3.2.4.2 信号传播 (INT)

PIPE 模块为 RTINT 和 INT 线路提供外部中断聚合和仲裁。这允许许多信号被归类为实时中断 (RTINT) 或低优先级中断 (INT)，然后在传递到 CPU 的 RTINT 或 INT 中断线路之前进行优先级排序。

PIPE 有效地多路复用了单个 INT CPU 中断线路，以便能够按适当的顺序从多个传入的 INT 中断中接收。

该模块允许在信号到达 CPU 的 INT 线路之前启用和禁用 INT 信号。该模块还允许嵌套功能以及分类为 INT 或 RTINT 的其他中断。有关 PIPE 模块功能的详细信息，请参阅 *F29H85x 和 F29P58x 实时微控制器技术参考手册*。

3.2.4.3 堆栈 (INT)

与 NMI 或 RTINT 中断线路不同，INT 线路使用标准软件栈来进行背景保存和恢复。多个可用 CPU 栈中只有一个可用于 INT。该结构由外部 PIPE 模块中的 INTSP 寄存器进行配置。如果 INT 向量指向与不同栈（安全分配的栈）相关的错误链接，则会生成 NMI 故障。

如果当前栈指针未指向 INTSP，则任何挂起的 INT 都将保持挂起状态，直到栈指针指向所选的 INTSP 栈。

3.3 阻止中断的条件

某些 CPU 流水线条件会导致 CPU 具有不可中断的边界。这些条件会阻止中断进入，直到条件结束，从而有效地在保持时间内阻止中断。表 3-2 说明了这些情况：

表 3-2. 阻止中断的条件

条件描述	INT 已阻止	RTINT 已阻止	NMI 已阻止
指令包中的条件指令未全部完成	是		
不连续指令延迟时隙未完成			
分支、调用、返回等多周期指令未完成			
对于 CALL.PROT 指令：未执行调用目标上的第一条指令			
对于 RET.PROT 指令：未执行返回地址的第一条指令			
前面激活的中断的第一条指令已经进入 D2 阶段			
CPU“流水线就绪”未置为有效			
由于存储器 RD/WR 访问，CPU 流水线停滞			
由于指令缓冲区中没有指令，CPU 流水线停滞			
由于存在流水线风险，指令包在流水线的 D2 阶段停止，但指令包未准备好移动到流水线的 R1 阶段。			
在 DSTS.INTE 中禁用 LP 中断	是	否	
ATOMIC 指令计数器未完成	是		否

3.3.1 ATOMIC 计数器

C29x CPU 支持将 8 位值加载到内部计数器 (ISTS.ATOMIC COUNTER) 中的指令“**ATOMIC.REG #u8**”。每执行一个指令包，此计数器递减一。只要此原子计数器不为零，中断 (RTINT 或 INT) 就无法进入 CPU 流水线。因此，这个指令允许用户代码针对多达 256 个指令包来阻止中断。

对使用 ATOMIC 指令的限制：

- ATOMIC 指令不能位于任何不连续指令的延迟时隙中，也不能与分支指令并行执行。
- ATOMIC 指令不能与任何不连续指令并行。
- 背对背执行 ATOMIC 指令不能用于阻止超过最大计数的中断。
- 当 ATOMIC 计数不为零时执行 ATOMIC 指令会将 ATOMIC 计数器复位。
- 受保护的调用和返回会使 ATOMIC 计数器复位。
- ATOMIC 指令或计数器无法阻止 NMI。只要设置了 ISTS.NMIF 标志 (指示 NMI 事件已注册) ，就会读取 NMI 并复位 ATOMIC 计数器。

3.4 CPU 中断控制寄存器

本节将介绍控制中断相关功能的三种 CPU 寄存器。

3.4.1 中断状态寄存器 (ISTS)

中断状态寄存器 (ISTS) 包含各种中断标志、栈指针、当前链接和各种计数器的状态信息。

-	CURRLINK				-			INTSP				CURRSP			
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

ATOMIC {counter}								-	-	-	-	-	NMIF	RTINTF	INTF
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

当前链接 (CURRLINK)：所有资源 (包括存储器、外设、栈) 都与链接 ID 相关联。链接会划分 CPU 运行环境的边界。因此，D2 流水线阶段中指令包的代码源地址解析为相应代码的链接。此信息对于验证、更新和访问中断向量表中的权限至关重要。该信息对于与每个中断关联的配置设置也至关重要。因此，CURRLINK 寄存器提供当前链接。

栈指针 (CURRSP、INTSP) ：具有嵌入式虚拟化功能的 C29x CPU 具有多个栈。用户可以为 INT 分配特定的栈，但 RTINT 和 NMI 使用 RTINT 栈。当前栈指针指向 CPU 正在使用的栈，由 CURRSP 字段表示。低优先级中断 (INT) 选择使用的栈由 INTSP 字段表示。在 INTSP 与 CURRSP 匹配之前，INT 不会进入流水线。

ATOMIC 计数器 (ATOMIC) ：CPU 允许在一次延展中执行多达 256 个指令包，而不会被 RTINT 或 INT 中断。ATOMIC 执行的剩余指令包的数量反映在 ATOMIC 计数器中。如果 ATOMIC 计数器正在计数，CPU 不会拾取中断进行处理。NMI 不受 ATOMIC 计数器的影响，运算被停止，如果接收到 NMI，计数器被复位。有关 ATOMIC 计数器的更多详细信息，请参阅节 3.3.1。

中断标志 (INTF、RTINTF、NMIF) ：寄存独立的中断标志，包括 INTF、RTINTF 和 NMIF。只要一个相应的中断被置为 CPU 有效，这些标志就会被置位并在退出相应的 ISR 时被清除。如果有多个嵌套中断被 CPU 采用，则所有相应的标志都将被设置，并且仅在处理相应类别的所有中断时清除这些标志。

3.4.2 解码阶段状态寄存器 (DSTS)

解码阶段状态寄存器 (DSTS) 包含有关 CPU 的中断和链接状态的信息。下表突出显示了与中断操作相关的字段。软件使用此信息来构建可预测的优先级和安全行为。

-	RLINK				ISR_PRIORITY								INTS		INTE
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16

TA3	TA2	TA1	TA0	-	CLINK				DBGM	-	-	A.ZV	A.C	A.N	A.Z
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

返回链接 (RLINK)：这表示执行受保护返回的源的链接。

ISR 优先级 (ISR_PRIORITY)：如果 CPU 正在处理中断并且 INT 是 INT 或 RTINT，则中断的优先级反映在此字段中。这是一个 8 位寄存器字段。

中断状态 (INTS)：该字段跟踪 CPU 执行的状态，指定执行是在主环路中、在 NMI ISR 中、在 RTINT ISR 中还是在 INT ISR 中。外部 PIPE 模块也使用此字段来跟踪 CPU 的当前阶段，以决定何时可以将下一个就绪的 RTINT 或 INT 中断转发到 CPU。有关 INTS 字段状态值的详细信息，请参阅表 3-3。

表 3-3. INTS - 中断状态值

INTS[1]	INTS[0]	CPU 状态	注释
0	0	主代码	不在任何任务、中断或异常中
0	1	INT 处理程序	在正常的中断服务例程中
1	0	RTINT 处理程序	在实时中断服务例程中
1	1	NMI 处理程序	在 NMI 处理程序例程中

INT 使能 (INTE)：INT 使能位反映 CPU 是否可以接受 INT 中断。该位需要为 1 才能在 CPU 中接受下一个更高优先级的 INT（这允许 INT 嵌套）。接受 INT 后，该位会自动设置为 0，因此 ISR 代码需要显式将该位设置回 1 以启用 INT 中断嵌套。

调用者链接 (CLINK)：CLINK 字段表示发起调用以执行该函数的源的链接。这包括来自 ISR 的执行调用。

为了增强安全性，可以在给定的 ISR（或函数）中检查 CLINK 字段，以确定 CLINK 字段是否与预定义的链接匹配。如果链接不匹配，ISR（或函数）随后会自动退出。

3.4.3 与中断相关的栈寄存器

C29x CPU 具有三种类型的栈，每个栈都具有相关的指针。这些内容在表 3-4 中进行了简要概述。下面的部分提供了与中断相关的高优先级中断栈指针的详细信息。

表 3-4. C29x CPU 栈类型

栈类型	相关指针
正常软件栈	SECSPx，其中 x = 0 至 15
受保护的调用栈	PSP、WARNPSP、MAXPSP
RTINT 栈	ISP、WARNRTISP、MAXRTISP

RTISP（RTINT 栈指针）：它指向 NMI 和 RTINT 中断线路使用的栈。该栈受 SSU 保护。有关 RTINT 栈的更多详细信息，请参阅节 3.2.3 的“栈”子部分。

WARNRTISP 级别：该级别由安全软件代码进行预编程。如果来自 CPU 的 ISP 达到此级别，则外部 PIPE 模块停止向 CPU 发送 RTINT。这是为了减慢栈进度或过多的嵌套，这些嵌套可能会导致栈溢出。用户可根据所需的软件安全检查更新 WARNRTISP 级别。通常在复位后修改 WARNRTISP 级别。

MAXRTISP 级别：固定嵌套数（等于高优先级中断栈所允许的嵌套总数减一）。这是为了让 NMI 触发一个保留的中断栈空间，以防止栈溢出。当达到此级别时，PIPE 会产生故障，进而产生 NMI 以解决此严重情况。

3.5 中断嵌套

C29x CPU 的硬件级别支持嵌套。在 CPU 中断级别，三个非复位中断线路（NMIn、RTINTn、INTn）中有可能嵌套。中断线路可以嵌套在较低优先级中断线路的 ISR 内。因此 NMI 可以嵌套在 RTINT 或 INT 内。RTINT 可以嵌套在 INT 内。INT 不能嵌套其他中断线路中。但是，中断类型 RTINT 和 INT 内的额外嵌套可以使用 PIPE 模块。

下面详细介绍了 C29x CPU 上可用的嵌套（以及 PIPE 模块提供的扩展功能）。

NMI：当前运行的 NMI 内不能嵌套任何中断（包括其他 NMI）。只要设置了 ISTS.NMIF 标志（指示 NMI 事件已注册），就会读取 NMI 并复位 ATOMIC 计数器。

RTINT：NMIS 始终嵌套在 RTINT 内。这个嵌套不能使用 ATOMIC 指令停止。通过使用 PIPE 模块，优先级较高的 RTINT 可以嵌套在较低优先级的 RTINT 中。ATOMIC 指令可能会延迟嵌套 RTINT 的进入，直到 ATOMIC 计数器到期。

INT：NMIS 始终嵌套在 INT 内。这个嵌套不能使用 ATOMIC 指令停止。RTINT 始终嵌套在 INT 中，但 ATOMIC 指令可能会延迟嵌套 RTINT 的进入，直到 ATOMIC 计数器到期。使用 PIPE 模块，较高优先级的 INT 可以嵌套在较低优先级 INT 内。ATOMIC 指令可以延迟嵌套 INT（或 RTINT）的进入，直到 ATOMIC 计数器到期。

3.5.1 中断嵌套示例图

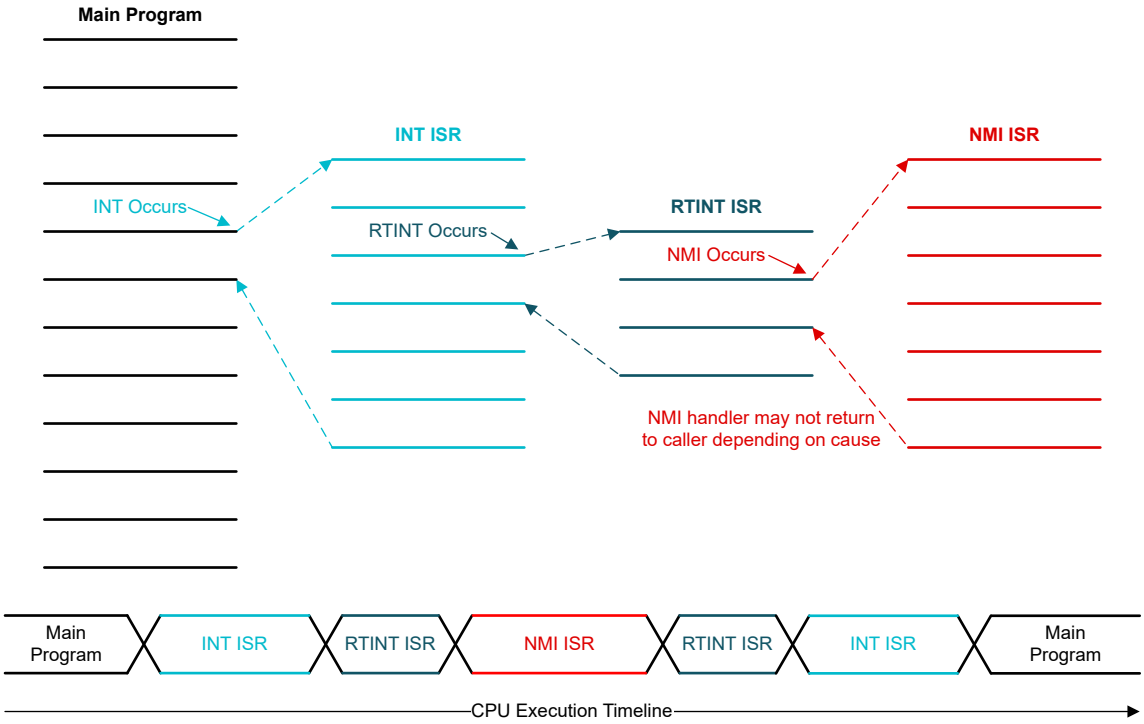

图 3-2. 中断嵌套示例图

3.6 安全性

C29x CPU 安全功能扩展到中断域以确保软件安全。本节介绍与 CPU 中断架构相关的安全特性。有关 C29x CPU 安全架构和 SSU（功能安全和信息安全单元）功能的详细信息，请参阅章节 5。

3.6.1 概述

C29x CPU 安全架构的核心是链接、栈和区域的概念。有关其中每项的工作方式的详细概述，请参阅章节 5。以下各节详细介绍 CPU 中断如何利用这些安全功能中的每一个。

3.6.2 链接

链接是一个所有权和访问权限的独特集合，由 ID 标记，用于资源分配和共享。链接 ID 绑定 CPU 资源，例如栈、存储器区域、外设实例访问、中断源和向量、DMA 通道和权限等。这可防止从另一个链接运行的黑客（或错误代码）访问资源。本章特别关注由链接提供的到特定中断源和向量的访问。

下面提供了用于中断操作的链接类型。

1. 所有者链接：每个中断线路和关联的中断向量都有一个所有者链接。
   - 此链接具有源事件的相关资源，例如外设、GPIO 或错误机制。
   - 所有者链接可以访问为特定事件提供服务所需的存储器、DMA 或其他资源。
   - 所有者链接具有对控制和状态寄存器的访问权限（读取标志、启用/禁用中断线路、清除标志或强制标志为高电平），并且还具有读取和清除溢出标志的权限。
   - 将 CPU 提供的当前链接 ID 与中断操作寄存器的所有者链接进行比较。
2. 引导链接：引导链接处理器件引导和初始化，包括中断。
   - 用户引导链接：用户引导链接没有访问 PIPE 或中断寄存器的特殊权限。
3. 安全根链接：这是器件安全代码的信任根，应可访问 PIPE 配置寄存器和向量表。配置寄存器会保存信息，例如中断线路的所有者链接、调用者链接、优先级、向量地址。
4. 调用者链接：调用者链接用于跨多个链接共享通用代码库。
   - 中断线路（如 RTINTn）的所有者链接可以在 ISR 中调用一个通用代码函数（来自另一个链接）。在这种情况下，将检查调用者链接是否为所有者链接。
   - 优先级等配置寄存器只能通过“安全根链接”进行更新。在这种情况下，所有者链接可以是“安全根链接”功能的调用者链接。

3.6.3 栈

C29x CPU 使用多个栈来确保不同进程之间的完整性和分离。每个链接在器件初始化时应映射一个关联的栈。多个链接可以共享一个栈，但多个栈不共享链接。下面列出了与 PIPE 和中断相关的栈，以及相应的安全特性：

- INT 栈：用户可为所有 INT 选择并分配一个栈。此栈是器件上可用的正常软件栈之一。向 CPU 发出的 INT 保持挂起状态，直到 CPU 返回此栈。通常，这应该是主进程的栈。
- RTINT 栈：这是专用栈用于 RTINT 和 NMI 的背景保存和恢复。对于任何用户代码而言，该栈均无法访问或查看，因此它包含了 ECC（错误校正代码）与寄存器。寄存器被置为零，这样就不能监测中断服务之前所发生的情况。高优先级中断栈上提供的特性包括：
  - *WARNRTISP 级别：*：该级别由安全软件代码进行预编程。如果来自 CPU 的 ISP 达到此级别，则外部 PIPE 模块停止向 CPU 发送 RTINT。这是为了减慢栈进度或过多的嵌套，这些嵌套可能会导致栈溢出。用户可根据所需的软件安全检查更新 WARNRTISP 级别。通常在复位后修改 WARNRTISP 级别。
  - *MAXRTISP 级别：*：固定嵌套数（等于高优先级中断栈所允许的嵌套总数减一）。这是为了让 NMI 触发一个保留的中断栈空间，以防止栈溢出。当达到此级别时，PIPE 会产生故障，进而产生 NMI 以解决此严重情况。

3.6.4 区域

可以组合多个栈和相应的链接以形成一个区域。可以通过链接和栈的配置反映构成此区域的项目。区域关联用于调试权限。

章节 4

寻址模式

本章介绍了 C29x CPU 的寻址模式并提供了示例。

4.1 寻址模式概述

C29x CPU 支持多种寻址模式，可提高执行速度并减小代码大小。

4.1.1 文档和实施

在本文中，使用寻址模式的指令通篇采用类似以下写法：“LD.32 Dx,ADDR1”。

在实际的汇编代码实现中，**字段**“ADDR1”被替换为实际**寻址模式**并使用替换的参数。例如：“*(Ax+#8)”。

这些寻址模式分为不同的**类型**。例如，*(Ax+#8) 的类型为“具有 #Immediate 偏移的指针寻址”。

下图直观地说明了在文档和实现中，字段、寻址模式和类型如何协同工作。两张图使用相同的**字段** (ADDR1)，但具有不同的**寻址模式**和寻址模式**类型**：

在图 4-1 中，ADDR1 被替换为特定寻址模式 ***(A15++#u8imm)**，这是**栈寻址**类型的寻址模式中可用的几种寻址模式之一。

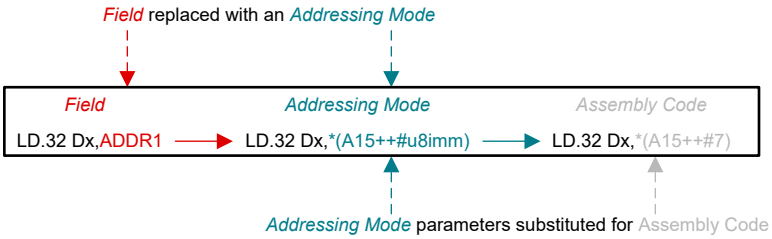

图 4-1. ADDR1 字段替换为栈寻址类型

在图 4-2 中，ADDR1 被替换为特定的寻址模式 ***(Ax+#u10imm)**，这是寻址模式的**具有 #Immediate 偏移的指针寻址**类型中可用的几种寻址模式之一。

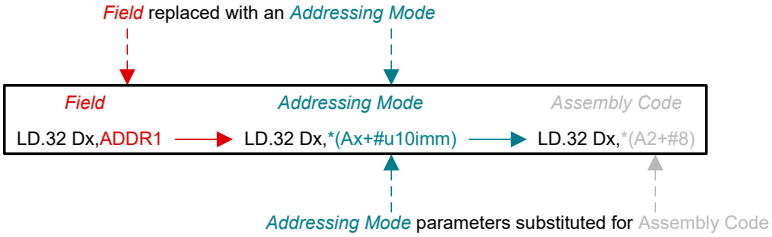

图 4-2. ADDR1 字段替换为具有 #Immediate 偏移的指针寻址

4.1.2 寻址模式类型列表

下面列出了器件本机可用的寻址模式类型。有关每种寻址模式的详细信息，请参阅节 4.4。

1. **直接寻址：**使用指令中提供的即时地址直接读取或写入 32 位存储器空间中的任何位置。
2. **具有 #Immediate 偏移的指针寻址：**对 32 位存储器空间中的任何位置进行间接读写，使用从其中一个寻址寄存器 A0 到 A14 的指针地址，以及指令中提供的可选即时偏移。
3. **具有指针偏移的指针寻址：**对 32 位存储器空间中的任何位置进行间接读写，其中指针地址（基址寄存器）来自寻址寄存器之一 A0 到 A14，以及由指令中的附加指针（索引寄存器）提供的偏移。
4. **具有 #Immediate 递增/递减的指针寻址：**使用指针地址从其中一个寻址寄存器 A0 至 A14 间接读取或写入访问 32 位存储器空间中的任何位置。立即对寄存器进行前置或后置递增或递减。
5. **具有指针递增/递减的指针寻址：**使用位于附加指针寄存器中的值应用对 32 位存储器空间中任何位置的间接读取或写入访问，其中指针地址从其中一个寻址寄存器 A0 到 A14，并应用寄存器的前置/后置递增或递减。
6. **栈寻址：**使用寻址寄存器 A15（专用栈指针 (SP)）中提供的地址间接读取或写入栈空间中的任何位置。

可以使用偏移和移位的不同组合来实现各种类型的寻址模式。所有可用的寻址模式以节 4.1.3 中的行的形式提供。

备注

寻址寄存器 **A15** 是专用的栈指针 (SP)。本文档中任何对“栈指针”或“SP”的引用均指对寄存器 A15 进行寻址。

4.1.2.1 其他寻址类型

可使用专门针对此任务的指令来执行另外两种寻址类型。这些指令用于获取地址，并在访问地址时修改/比较该地址。这样就可实现通常需要在一条指令中出现多条指令的操作。这样就不需要为此类功能使用专用寻址模式。将原生寻址模式支持与这些指令进行比较时，对性能没有影响。

1. **循环寻址：**循环指针通常用于实现有限脉冲响应 (FIR)、最小均方根 (LMS) 或卷积滤波器。
2. **位反向寻址：**反向寻址通常用于对快速傅里叶变换 (FFT) 和类似算法的数据重新排序。

4.1.3 寻址模式汇总

表 4-1 总结了所有受支持的寻址模式和各种形式。

表 4-1. 可用寻址模式

操作码字段	助记符	简写	地址生成
直接寻址			
DIRM	*(0:#u32imm)	@u32imm	addr = #u32imm
具有 #Immediate 偏移的指针寻址：（Ax = A0 至 A14[2]，Az = A4 至 A7）			
DIRM	*(Ax+#u28imm)	*Ax[#u28imm]	addr = Ax + #u28imm（#u28imm = 0 至 256MB 范围）
ADDR1	*(Ax+#u10imm)	*Ax[#u10imm]	addr = Ax + #u10imm（#u10imm = 0 至 1KB 范围）
ADDR1	*(Ax+#u10imm<<2)	*Ax[#u10imm]	addr = Ax + #u10imm<<2（#u10imm << 2 = 0 至 4KB 范围，4B 步长）
ADDR3	*(Ax+#u8imm<<2)	*Ax[#u8imm]	addr = Ax + #u8imm<<2（#u8imm << 2 = 0 至 1KB 范围，4B 步长）
ADDR2	*Az	*Az	addr = Az
具有指针偏移的指针寻址：（Ax = A0 至 A14[2]、Aj = A0 至 A14、Ak = A0 至 A3、Az = A4 至 A7）			
ADDR1	*(Ax+Ak<<#u2imm)	*Ax[Ak]	addr = Ax + Ak << #u2imm（#u2imm = 0、1、2、3）
ADDR1	*(Aj=(Ax+Ak<<#u2imm))	*Aj=Ax[Ak]	addr = Ax + Ak << #u2imm, Aj = addr（#u2imm = 0、1、2、3）
ADDR2	*(Az+A0<<#scale)	*Az[A0]	addr = Az + A0 << (0/1/2/3)[1]
ADDR2	*(Az+A1<<#scale)	*Az[A1]	addr = Az + A1 << (0/1/2/3)[1]
具有 #Immediate 递增/递减的指针寻址：（Ax = A0 至 A14[2]，Az = A4 至 A7）			
ADDR1	*(Ax++#u8imm)	*Ax++[#u8imm]	addr = Ax, Ax = Ax + #u8imm（#u8imm = 0 至 255 范围）
ADDR1	*(Ax--#n8imm)	*Ax--[#n8imm]	addr = Ax, Ax = Ax - #n8imm（#n8imm = 1 至 256 范围）
ADDR1	*(Ax-=#n8imm)	*Ax-=[#n8imm]	Ax = Ax - #n8imm, addr = Ax（#n8imm = 1 至 256 范围）
ADDR2	*(Az++#size)	*Az++	addr = Az，Az = Az + (1/2/4/8) (#size = 1,2,4,8)[1]
ADDR2	*(Az--#size)	*Az--	addr = Az，Az = Az – (1/2/4/8) (#size = 1,2,4,8)[1]
ADDR2	*(Az-=#size)	*--Az	Az = Az – (1/2/4/8)，addr = Az (#size = 1,2,4,8)[1]
具有指针递增/递减的指针寻址：（Ax = A0 至 A14[2]、Ak = A0 至 A3、Az = A4 至 A7）			
ADDR1	*(Ax+#u7imm)++Ak	*Ax[#u7imm]++Ak	addr = Ax + #u7imm，Ax = Ax + Ak（#u7imm = 0 至 128）
ADDR2	*(Az++A0)	*Az++A0	addr = Az，Az = Az + A0
ADDR2	*(Az++A1)	*Az++A1	addr = Az，Az = Az + A1
栈寻址：(A15 = SP)			
ADDR1	*(A15-#n13imm)	*A15-[#n13imm]	addr = A15 - #n13imm（#n13imm = 1 至 8192）
ADDR1	*(A15++#u8imm)	*A15++[#u8imm]	addr = A15，A15 = A15 + #u8imm（#u8imm = 0 至 255）
ADDR1	*(A15-=#n8imm)	*A15-=[#n8imm]	A15 = A15 - #n8imm，addr = A15（#n8imm = 1 至 256）

(1) ADDR2 操作码字段模式不指定增量步长（“#size”）或缩放量（“#scale”）。这由 CPU 硬件根据指令所访问的字大小自动执行。有关更多详细信息，请参阅节 4.2。

(2) Ax[0-14] 寻址字段可支持 A15 寄存器，但这是栈指针 (SP) 寄存器，对于某些寻址模式，该操作对 SP 无效，因此不能使用寻址模式

4.2 寻址模式字段

本节说明了每条指令中不同的寻址模式是如何表示的。

术语解释

对于使用寻址模式的指令，本文档使用四个不同的“**字段**”：

- ADDR1
- ADDR2
- ADDR3
- DIRM

这四个字段是实际寻址模式的占位符。这四个字段根据在指令中对它们进行编码所需的位数进行分隔（例如，ADDR1 使用 16 位，ADDR2 使用 5 位）。

在实际汇编代码中，用户或编译器必须替换字段以获得所需的寻址模式。例如，在文档中，使用“field”名称 ADDR1。但在汇编代码中，可将 ADDR1 替换为实际寻址模式。

“LD.32 Ax,**ADDR1**”（字段）

变为

“LD.32 A8,***(A4+#0x4)**”（寻址模式）。

这是将 ADDR1 字段中可用的 16 位转换为实际寻址模式（寻址模式的“具有 #Immediate 偏移的指针寻址”类型）的唯一方法。

以下各小节将逐一介绍 4 个字段中的每一个字段、可用的寻址模式，以及这些寻址模式的编码。还有一个小节介绍寻址模式中使用的一些额外字段。

4.2.1 ADDR1 字段

这是一个用于地址间接编码的 16 位字段，可在所有“指针寻址”和“栈寻址”模式下使用。

表 4-2 展示了使用 16 位对地址进行编码的各种方式。

表 4-2. ADDR1 字段编码

ADDR1 字段：（Ax = A0 至 A14，Aj = A0 至 A14，Ak = A0 至 A3）																		
助记符 [2]	简写	地址生成	47	46	45	43	42	41	40	39	38	37	36	35	34	33	32	31
助记符	简写	地址生成	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
*(Ax+#u10imm)	*Ax[#u10imm]	addr = Ax + #u10imm（#u10imm = 0 至 1KB 范围）	0	0	#u10imm										Ax[0-14] [1]			
*(Ax+#u10imm<<2)	*Ax[#u10imm]	addr = Ax + #u10imm<<2（#u10imm << 2 = 0 至 4KB 范围，4B 步长）	0	1	#u10imm										Ax[0-14] [1]			
*(Ax+#u7imm)++Ak	*Ax[#u7imm]++Ak	addr = Ax + #u7imm，Ax = Ax + Ak（#u7imm = 0 至 128）	1	0	0	#u7imm							Ak[0-3]		Ax[0-14] [1]			
*(A15-#n13imm)	*A15-[#n13imm]	addr = A15 - #n13imm（#n13imm = 1 至 8192）	1	0	1	#n13imm												
*(Ax++#u8imm)	*Ax++[#u8imm]	addr = Ax, Ax = Ax + #u8imm（#u8imm = 0 至 255 范围）	1	1	0	0	#u8imm								Ax[0-14] [1]			
*(Ax--#n8imm)	*Ax--[#n8imm]	addr = Ax, Ax = Ax - #n8imm（#n8imm = 1 至 256 范围）	1	1	0	1	#n8imm								Ax[0-14] [1]			
*(Ax-=#n8imm)	*Ax-=[#n8imm]	Ax = Ax - #n8imm, addr = Ax（#n8imm = 1 至 256 范围）	1	1	1	0	#n8imm								Ax[0-14] [1]			
*(Ax+Ak<<#u2imm)	*Ax[Ak]	addr = Ax + Ak << #u2imm（#u2imm = 0、1、2、3）	1	1	1	1	#u2imm		1	1	1	1	Ak[0-3]		Ax[0-14] [1]			
*(Aj=(Ax+Ak<<#u2imm))	*Aj=Ax[Ak]	addr = Ax + Ak << #u2imm, Aj = addr（#u2imm = 0、1、2、3）	1	1	1	1	#u2imm		Aj[0-14]				Ak[0-3]		Ax[0-14] [1]			

(1) Ax[0-14] 寻址字段可支持 A15 寄存器，但这是栈指针 (SP) 寄存器，对于某些寻址模式，该操作对 SP 无效，因此不能使用寻址模式。
(2) 数据移动操作有两个 ADDR1 字段，所有其他操作只有一个 ADDR1 字段。除数据移动操作外，ADDR1 字段位于指令操作码的 [31:16] 位。

以下是可以使用 ADDR1 字段的指令：

ADD.32、ADD.S16、ADD.S8、AND.16、AND.8、AND.U16、AND.U8、ANDOR.B0、ANDOR.W0、LD.32、LD.64、LD.B0、LD.B1、LD.B2、LD.B3、LD.S16、LD.S8、LD.U16、LD.U8、LD.W0、LD.W1、MV.16、MV.32、MV.64、MV.8、MV.U16、MV.U8、OR.16、OR.8、RET{D}、S16TOF、ST.16、ST.32、ST.64、ST.8、ST.B0、ST.B1、ST.B2、ST.B3、ST.W0、ST.W1、SUB.32、SUB.S16、SUB.S8、SUBR.32、SUBR.S16、SUBR.S8、U16TOF、XOR.16、XOR.8

示例：

```
; Load the 32-bit value in ADDR1 into Mx, using a base address + offset
; (#u7imm) and then post increment by Ak (Ak is number of bytes to increment)
; NOTE: make sure 32-bit alignment of base address (Ax) and offset
LD.32 Mx,ADDR1              ; field
LD.32 Mx,*(Ax+#u7imm)++Ak   ; addressing mode
LD.32 M1,*(A14+#100)++A2    ; actual assembly code

; OR #x16 with the address pointed to by ADDR1, and store the result
; into the location pointed to by ADDR1. Then post decrement the Ax register
; by the #n8imm value
; NOTE: make sure 16-bit alignment of base address (Ax) and offset
OR.16 ADDR1,#x16            ; field
OR.16 *Ax--[#n8imm],#x16    ; addressing mode
OR.16 *A3--[#70],#50110     ; actual assembly code
```

4.2.2 ADDR2 字段

这是一个用于地址间接编码的 5 位字段，可在所有“指针寻址”模式下使用。

表 4-3 展示了使用 5 位对地址进行编码的各种方式。

表 4-3. ADDR2 字段编码

ADDR2 字段：(Az = A4 到 A7)							
助记符	简写	地址生成	4	3	2	1	0
*(Az++A0)	*Az++A0	addr = Az，Az = Az + A0	0	0	0	Az[4-7]	
*(Az++A1)	*Az++A1	addr = Az，Az = Az + A1	0	0	1	Az[4-7]	
*(Az+A0<<#scale)	*Az[A0]	addr = Az + A0 << (0/1/2/3) (1)	0	1	0	Az[4-7]	
*(Az+A1<<#scale)	*Az[A1]	addr = Az + A1 << (0/1/2/3) (1)	0	1	1	Az[4-7]	
*Az	*Az	addr = Az	1	0	0	Az[4-7]	
*(Az++#size)	*Az++	addr = Az，Az = Az + (1/2/4/8) (#size = 1,2,4,8) (1)	1	0	1	Az[4-7]	
*(Az--#size)	*Az--	addr = Az，Az = Az – (1/2/4/8) (#size = 1,2,4,8) (1)	1	1	0	Az[4-7]	
*(Az-=#size)	*--Az	Az = Az – (1/2/4/8)，addr = Az (#size = 1,2,4,8) (1)	1	1	1	Az[4-7]	

备注

ADDR2 操作码字段模式不指定增量步长（“#size”）或缩放量（“#scale”）。这由 CPU 硬件根据指令所访问的字大小自动执行。

- 字节访问按 #size=1 递增/递减，或按 #scale=0 缩放（乘以 1）
- 16 位访问按 #size=2 递增/递减，或按 #scale=1 缩放（乘以 2）
- 32 位访问按 #size=4 递增/递减，或按 #scale=2 缩放（乘以 4）
- 64 位访问按 #size=8 递增/递减，或按 #scale=3 缩放（乘以 8）

以下是可以使用 ADDR2 字段的指令：

AND.32、ANDOR、LD.32、LD.64、MV.16、MV.32、MV.64、MV.8、OR.32、ST.16、ST.32、ST.64、ST.8、XOR.32

示例：

```
; Register XMx is loaded with the 64-bit word at the memory location
; addressed using the ADDR2 addressing mode. This address is fetched from Az.
LD.64 XMx,ADDR2             ; field
LD.64 XMx,*Az               ; addressing mode
LD.64 XM4,*A4               ; actual assembly code

; The 8-bit immediate value specified is stored at the memory location
; addressed using the ADDR2 addressing mode. The address is fetched from Az,
; which is then post-decremented by the amount in #size. The #size is
; automatically chosen by the CPU to be 1 since the word size accessed by
; this instruction is 8-bit.
ST.8 ADDR2,#0x0B            ; field
ST.8 *(Az--#size),#0x0B     ; addressing mode
ST.8 *(Az--#1),#0x0B        ; actual assembly
```

4.2.3 ADDR3 字段

这是一个 12 位字段，用于对仅用于“具有 #Immediate 偏移的指针寻址”的地址进行间接编码。

表 4-4 展示了使用 12 位对地址进行编码的各种方式。

表 4-4. ADDR3 字段编码

ADDR3 字段：（Ax = A0 至 A14）														
助记符	简写	地址生成	11	10	9	8	7	6	5	4	3	2	1	0
*(Ax+#u8imm<<2)	*Ax[#u8imm]	addr = Ax + #u8imm<<2（#u8imm << 2 = 0 至 1KB 范围，4B 步长）	Ax[0-14] [1]				#u8imm							

(1) Ax[0-14] 寻址字段可支持 A15 寄存器，但这是栈指针 (SP) 寄存器，对于某些寻址模式，该操作对 SP 无效，因此不能使用寻址模式。

以下是可以使用 ADDR3 字段的指令：

MV.32

示例：

```
; The 32-bit content at the memory location addressed using the ADDR3
; addressing mode, ADDR3_x, is copied to the memory location addressed using
; the ADDR3 addressing mode, ADDR3_y. Both ADDR3 fields use the same
; addressing mode "*(Ax+#u8imm<<2)", which calculates the address using a
; base pointer added with an 8-bit immediate (#u8imm) that is multiplied by 4
; (#u8imm<<2). NOTE: The base address must be 32-bit aligned.
MV.32 ADDR3_y,ADDR3_x                    ; field
MV.32 *(Ax+#u8imm<<2),*(Ax+#u8imm<<2)    ; addressing mode
MV.32 *(A0+#4<<2),*(A1+#8<<2)            ; actual assembly
```

4.2.4 DIRM 字段

DIRM 字段

这是一种 33 位编码，用于对仅用于“直接寻址”和“具有 #lmmediate 偏移的指针寻址”的地址进行直接和间接编码。

表 4-5 展示了使用 12 位对地址进行编码的各种方式。

表 4-5. DIRM 字段编码

DIRM 字段：(Ax = A0 至 A14)						
助记符		地址生成	0	31:20	19:16	47:32
*(0:#u32imm)	@u32imm	addr = #u32imm	0	#u32imm		
*(Ax+#u28imm)	*Ax[#u28imm]	addr = Ax + #u28imm (#u28imm = 0 至 256MB 范围)	1	#u28imm (低 12 位)	Ax[0-14] [1]	#u28imm (高 16 位)

(1) Ax[0-14] 寻址字段可支持 A15 寄存器，但这是栈指针 (SP) 寄存器，对于某些寻址模式，该操作对 SP 无效，因此不能使用寻址模式。

以下是可以使用 DIRM 字段的指令：

LD.32、LD.64、LD.B0、LD.B1、LD.B2、LD.B3、LD.S16、LD.S8、LD.U16、LD.U8、LD.W0、LD.W1、S16TOF、ST.32、ST.64、ST.B0、ST.B1、ST.B2、ST.B3、ST.W0、ST.W1、U16TOF

示例：

```
; Bits [7:0] of register Ax are loaded with the 8-bit value at the memory
; location addressed using the DIRM addressing mode. DIRM is supplied with a
; 32-bit unsigned immediate value found in parklSine:
; parklSine = 0x00008000
LD.B0 Ax,DIRM               ; field
LD.B0 Ax,@u32imm            ; addressing mode
LD.B0 A8,@parklSine         ; actual assembly

; The upper 16-bit content of register Ax is stored at the memory location
; addressed using the DIRM addressing mode. The DIRM field is replaced with
; the "*(Ax+#u28imm)" addressing mode, where the address is found using
; base pointer Ax and the #u28imm immediate value.
ST.W1 DIRM,Ax               ; field
ST.W1 *(Ax+#u28imm),A10     ; addressing mode
ST.W1 *(A3+#0x4),A10        ; actual assembly
```

4.2.5 其他字段

除了寻址模式字段之外，还有在实际寻址模式中使用的 #immediate 字段，例如“*(Ax+#u10imm)”寻址模式中的“#u10imm”。其中大多数 #immediate 字段（也称为常量）显而易见（例如，#u10imm 是无符号的 10 位直接部分）。

但是，为了清晰起见，将使用表对两个负 #immediate 字段进行详细说明：

#n13imm 字段

#n13imm 字段是 13 位负偏移 #immediate，用于“*(A15-#n13imm)”寻址模式。该寻址模式是可用的 ADDR1 字段之一（需要 16 位进行编码），属于“栈寻址”类型。

使用此 #immediate 提供一个负 13 位值，并且位 13 至 31 用 1 填充，以创建 32 位负偏移常量。

备注

位 13 至位 31 用 1 填充，以创建一个 32 位负偏移常量，然后将其添加到寻址寄存器。

表 4-6. #n13imm 字段编码

12	11	10	9	8	7	6	5	4	3	2	1	0	编码值	符号扩展值
1	1	1	1	1	1	1	1	1	1	1	1	1	1	-1
1	1	1	1	1	1	1	1	1	1	1	1	0	2	-2
...														
1	0	0	0	0	0	0	0	0	0	0	0	1	4095	-4095
1	0	0	0	0	0	0	0	0	0	0	0	0	4096	-4096
0	1	1	1	1	1	1	1	1	1	1	1	1	4097	-4097
...														
0	0	0	0	0	0	0	0	0	0	0	0	1	8191	-8191
0	0	0	0	0	0	0	0	0	0	0	0	0	8192	-8192

#n8imm 字段

#n8imm 字段是 8 位负偏移 #immediate，用于以下寻址模式：

- n
- *(Ax--#n8imm)，即寻址模式类型为“具有 #immediate 递增/递减的指针寻址”
- *(Ax-=#n8imm)，即寻址模式类型为“具有 #Immediate 递增/递减的指针寻址”
- *(A15-=#n8imm)，即寻址模式类型“栈寻址”

这些寻址模式都是可用 ADDR1 字段的一部分（所有这些都需要 16 位进行编码）。

使用此 #immediate 提供一个负 8 位值，并且位 8 至 31 用 1 填充，以创建 32 位负偏移常量。

备注

位 8 至位 31 用 1 填充，以创建一个 32 位负偏移常量，然后将其添加到寻址寄存器。

表 4-7. #n8imm 字段编码

7	6	5	4	3	2	1	0	编码值	符号扩展值
1	1	1	1	1	1	1	1	1	-1
1	1	1	1	1	1	1	0	2	-2
...									
1	0	0	0	0	0	0	1	127	-127
1	0	0	0	0	0	0	0	128	-128
0	1	1	1	1	1	1	1	129	-129
...									
0	0	0	0	0	0	0	1	255	-255
0	0	0	0	0	0	0	0	256	-256

4.3 对齐和流水线注意事项

本节介绍寻址对齐的要求以及寻址模式的流水线注意事项。

4.3.1 对齐

所有数据访问都与最近的字大小对齐。这由正在访问的存储器或外设强制执行。

这意味着所有数据访问都需要执行以下操作：

- 基指针地址必须与数据字宽度对齐。
- 任何偏移或递增/递减大小都必须是数据大小的倍数。
- 最终地址（应用任何偏移或递增/递减的基指针）必须与数据字宽度对齐。

> **小心**
> 编译器会根据字大小自动进行适当的偏移索引编制和缩放。但是，如果从存储器位置加载基指针，编译器会假定内容正确对齐。如果内容未对齐，并且生成的特定字大小地址未对齐，则会生成 CPU 寻址故障。

有关基地址的示例：基指针地址必须与数据字宽度对齐。因此，如果使用诸如“LD.64”的 64 位（8 字节）数据指令，基址必须与 64 位字边界对齐。因此，二进制地址的最后三位数字必须为 0，因为这意味着该值可以被 8 整除。因此“LD.32 D2,*(0:**#0xF8**)”可以有效（因为在二进制中，这是 0b1111 1**000**），但“LD.32 D2,*(0:**#0xF9**)”不能有效（因为在二进制中，这是 0b1111 1**001**）。

有关偏移的示例：使用的任何偏移值（以字节为单位）都必须是指令数据大小的倍数。因此，如果使用诸如“LD.32”的 32 位（4 字节）数据指令，则偏移必须是 4 的倍数。因此，“LD.32 D2,*(A2 + **#4**)”可能有效，但“LD.32 D2,*(A2 + **#5**)”可能无效。这些指令还要求基指针对齐。

下面提供了一些有关正确和错误对齐的其他示例：

```
MV.32       A2,#ArrayX          ; Assume that the array is aligned
                                ; to a 64-bit word boundary for this example.
; CORRECT Examples:
; Pointer Addressing With #Immediate Offset Examples
LD.B0       D0,*(A2+#9)     ; Byte offset can be any value
LD.U16      D1,*(A2+#10)    ; 16-bit offset can only be a multiple of 2 bytes
LD.32       D2,*(A2+#4)     ; 32-bit offset can only be a multiple of 4 bytes
LD.64       XD4,*(A2+#16)   ; 64-bit offset can only be a multiple of 8 bytes
; Scaled values (left shift or multiplied values)
LD.U16      D1,*(A2+#1<<1)  ; 16-bit offset can only be a multiple of 2 bytes
LD.U16      D1,*(A2+#3<<1)  ; 16-bit offset can only be a multiple of 2 bytes
LD.64       XD4,*(A2+#2<<3) ; 64-bit offset can only be a multiple of 8 bytes
; Pointer Addressing with #Immediate Increment/Decrement Examples
LD.B0       D0,*(A2++#9)    ; Byte offset can be any value
LD.U16      D1,*(A2++#10)   ; 16-bit offset can only be a multiple of 2 bytes
LD.32       D2,*(A2++#4)    ; 32-bit offset can only be a multiple of 4 bytes
LD.64       XD4,*(A2++#24)  ; 64-bit offset can only be a multiple of 8 bytes

; INCORRECT Examples:
LD.U16      D1,*(A2++#5)    ; INCORRECT: offset can only be a multiple of 2
LD.U16      D1,*(A2+#3<<0)  ; INCORRECT: offset can only be a multiple of 2
LD.64       XD4,*(A2+#10)   ; INCORRECT: offset can only be a multiple of 8
; If ArrayX is not aligned to a 32-bit boundary and LD.32 is called,
; then a CPU addressing fault is generated.
```

备注

ADDR2 操作码字段模式不指定增量步长（“#size”）或缩放量（“#scale”）。这由 CPU 硬件根据指令所访问的字大小自动执行。

- 字节访问按 #size=1 递增/递减，或按 #scale=0 缩放（乘以 1）
- 16 位访问按 #size=2 递增/递减，或按 #scale=1 缩放（乘以 2）
- 32 位访问按 #size=4 递增/递减，或按 #scale=2 缩放（乘以 4）
- 64 位访问按 #size=8 递增/递减，或按 #scale=3 缩放（乘以 8）

4.3.2 流水线注意事项

虽然 C29x CPU 实现了一个完全受保护的流水线，但需要考虑一些注意事项：

- 在一个指令包中最多可以并行进行两个加载和一个存储。这可能包括修改 Ax 寻址寄存器。一个 Ax 寄存器可以被并行读取多次，但在单个指令包中不能被并行写入多次。
- 如果任何汇编代码在同一个指令包内有多条指令修改同一目标寄存器，汇编语言工具会对这种情况进行标记。

下面提供了一个有效代码示例，其中在单个指令包中包含两个加载和一个存储指令：

```
LD.32       D0,*A2+A0       ; Use A0 as an index from A2
||LD.32     D1,*A2+A1       ; Use A1 as an index from A2
||ST.32     *(A2-=#4),D3    ; Pre-Decrement A2
||ADD       A0,A0,#6        ; Add #6 to A0 register
||SUB       A1,A1,#10       ; Sub #10 from A1 register
; each Ax register is only modified once here.
```

4.4 寻址模式类型

本节详细介绍了 C29x CPU 中可用的每种寻址模式类型。

4.4.1 直接寻址

借助“直接寻址”类型，可以使用指令中提供的即时地址直接读取或写入 32 位存储器空间中的任何位置。

典型用例用于访问固定地址位置（例如外设寄存器）或位于固定存储器位置（在构建时）的变量。

该类型的缺点包括此寻址模式仅适用于 48 位指令，因此大量使用此类寻址模式的代码会占用更多空间。

该类型的优点包括该寻址模式不需要任何寻址指针。如果所访问的值很少并随机分散在用户程序中，这种类型的寻址模式比尝试初始化指针并使用指针寻址更高效。

示例

```
LD.U16 D0,@ADC2.ResultReg5  ; Load register D0 with the location of
                            ; ADC2 peripheral result register 5
```

4.4.2 指针寻址

4.4.2.1 具有 #Immediate 偏移的指针寻址

借助“具有 #Immediate 偏移的指针寻址”类型，可以对 32 位存储器空间中任何位置进行间接读取或写入访问，使用从其中一个寻址寄存器 A0 到 A14 的指针地址，以及指令中提供的可选即时偏移。

使用完整的 32 位无符号加法运算将即时偏移添加到基址寄存器中。如果值溢出，该值将绕回。

典型的用例是以任意随机顺序多次索引到给定的数据数组或外设中。此类型中的每种寻址模式都可根据特定用途进行定制：

*(Ax+#u28imm)	用于实现与位置无关的代码，或在访问非常大的数据数组时。
*(Ax+#u10imm)	用于访问 1KB 或更小的数据数组。
*(Ax+#u10imm<<2)	用于访问外设寄存器（外设的寄存器范围为 4KB 或 4KB 的倍数，并在 32 位字边界上对齐）。
*(Ax+#u8imm<<2)	用于在两个小于 1KB 的 32 位数据数组之间移动多个数据条目。仅用于一条数据移动指令，使该功能采用紧凑的 32 位指令。

备注

所有数据访问必须与最近的字大小对齐。有关对齐的更多详细信息和注意事项，请参阅节 4.3.1。

4.4.2.2 具有指针偏移的指针寻址

借助“具有指针偏移的指针寻址”类型，可以对 32 位存储器空间中的任何位置进行间接读写，其中指针地址（基址寄存器）来自寻址寄存器之一 A0 到 A14，以及由指令中的附加指针（索引寄存器）提供的偏移。

利用该结构，可以使用变量索引轻松访问数据数组。这方面的一个示例可以通过 32 位整数组来演示。如果需要数组中的第八个 int，则可以按如下方式访问该元素：

```
; Because the array is of type int, each element is 4 bytes (32 bits) long.
;    So the index must be multiplied by 4 (which is the same as <<2)
; Starting parameters:
;    int arr[7] = 12
;    A2 = arr (base address)
;    A0 = i (index) = 7 (the eigth int in the array)

LD.32 D0,*(A2+A0<<2)    ; D0 = arr + (7<<2 byte offset)

; Result:
;    D0 = 12
```

- 8 位访问不需要将索引“i”相乘（无需移位）。16 位访问要求将“i”乘以 2 (<<1)，以实现 2 字节对齐。
- 32 位访问要求将“i”乘以 4 (<<2)，以实现 4 字节对齐。
- 64 位访问要求将“i”乘以 8 (<<3)，以实现 8 字节对齐。

由寄存器和移位提供的偏移使用完整的 32 位无符号加法运算添加到基址寄存器中。如果值溢出，该值将绕回。

这样就可以绕回负索引值：

```
; Starting parameters:
;    A2 = arr =  8 = 0x0000 0008 (base address at 8th byte in memory space)
;    A0 = i   = -1 = 0xFFFF FFFF (index at -1)
*(A2+A0) = 8 + (-1) = 7th byte in memory space
```

备注

所有数据访问必须与最近的字大小对齐。有关对齐的更多详细信息和注意事项，请参阅节 4.3.1。

4.4.2.3 具有 #Immediate 递增/递减的指针寻址

借助“具有 #Immediate 递增/递减的指针寻址”类型，可以使用指针地址从其中一个寻址寄存器 A0 至 A14 间接读取或写入访问 32 位存储器空间中的任何位置。立即对寄存器进行前置或后置递增或递减。

根据 #immediate 值提供的递增偏移大小会使用完整的 32 位无符号加法运算添加到基址寄存器中。如果值溢出，该值将绕回。

根据 #immediate 值提供的递减偏移大小，并使用完整的 32 位无符号子运算将其添加到基址寄存器中。如果该值下溢，则会绕回。

备注

所有数据访问必须与最近的字大小对齐。有关对齐的更多详细信息和注意事项，请参阅节 4.3.1。

备注

ADDR2 操作码字段模式不指定增量步长（“#size”）或缩放量（“#scale”）。这由 CPU 硬件根据指令所访问的字大小自动执行。

- 字节访问按 #size=1 递增/递减，或按 #scale=0 缩放（乘以 1）
- 16 位访问按 #size=2 递增/递减，或按 #scale=1 缩放（乘以 2）
- 32 位访问按 #size=4 递增/递减，或按 #scale=2 缩放（乘以 4）
- 64 位访问按 #size=8 递增/递减，或按 #scale=3 缩放（乘以 8）

4.4.2.4 具有指针递增/递减的指针寻址

借助“具有 #Immediate 递增/递减的指针寻址”类型，可以使用位于附加指针寄存器中的值应用对 32 位存储器空间中任何位置的间接读取或写入访问，其中指针地址从其中一个寻址寄存器 A0 到 A14，并应用寄存器的前置/后置递增或递减。

此类型中的寻址模式之一“*(Ax+#u7imm)++Ak”允许指针递增/递减以及偏移。这在访问值接近变量索引的代码中非常有用。C 语言和生成的汇编代码提供了这方面的一个示例：

C 代码：

```
For (i=0; i<N; i++)
{
    ArrayY[i]   = ArrayX[i] + ArrayX[i+1];
    ArrayY[i+1] = ArrayX[i] - ArrayX[i+1];
}
```

生成的汇编代码：

```
; Initialize ArrayX and ArrayY Pointers and i:
MV          A0,#4                   ; A0 = i = 4 = increment step size
MV          A2,#ArrayX              ; A2 = ArrayX base address
MV          A3,#ArrayY              ; A3 = ArrayY base address
...
; This code is repeated N times:
LD.32       D0,*(A2 + #0)           ; D0 = ArrayX[i]
||LD.32     D1,*(A2 + #1*4)++A0     ; D1 = ArrayX[i+1], A2 = A2 + A0
ADD         D2,D1,D0                ; D2 = ArrayX[i] + ArrayX[i+1];
||SUB       D3,D1,D0                ; D3 = ArrayX[i] - ArrayX[i+1];
ST.32       *(A3 + #0),D2           ; ArrayY[i] = D2
ST.32       *(A3 + #1*4)++A0        ; ArrayY[i+1] = D3, A3 = A3 + A0
```

根据 #immediate 值提供的递增偏移大小会使用完整的 32 位无符号加法运算添加到基址寄存器中。如果值溢出，该值将绕回。

这种绕回可用于实现递减索引。例如：

```
; Starting parameters:
;    A2 = arr =  8 = 0x0000 0008 (base address at 8th byte in memory space)
;    A0 = i   = -1 = 0xFFFF FFFF (index at -1)
*(A2+A0) = 8 + (-1) = 7th byte in memory space
```

备注

所有数据访问必须与最近的字大小对齐。有关对齐的更多详细信息和注意事项，请参阅节 4.3.1。

4.4.3 栈寻址

借助“栈寻址”类型，可以使用寻址寄存器 A15（专用栈指针 (SP)）中提供的地址读取或写入栈空间中的任何位置。

以下是与栈指针相关的关键信息列表，有助于理解这些寻址模式：

- 寻址寄存器 A15 是专用的栈指针。
- 栈从低位地址增长到高位地址。
- 栈指针始终指向栈顶部的下一个空位置。
- 栈指针必须始终与 64 位字边界对齐。如果栈指针未对齐，中断、调用和返回操作会生成故障。
- 如果栈指针与所访问的字大小未对齐，这也会生成故障。

分配栈空间并访问栈上的值时，建议的过程如下：

- 以 8 字节（64 位）为增量分配栈空间
- 使用 ***(A15-#n13imm)** 寻址模式访问栈内容（所有访问必须与所访问的字大小对齐）
- 完成后，取消分配栈空间（递减 8 字节）

例如：程序需要为以下各项分配空间：

- 1* 64 位值
- 3 * 32 位值
- 1 * 16 位值
- 3 * 8 位值（字节）

考虑到对齐，要分配的总字节数可以是 32（这是高于所需 25 字节的最接近的 64 位地址）：

```
64-bit  +8      8  bytes total      requires 8 bytes allocated
32-bit  +4      12 bytes total      requires 16 bytes allocated
32-bit  +4      16 bytes total
32-bit  +4      20 bytes total      requires 24 bytes allocated
16-bit  +2      22 bytes total
```

```
8-bit   +1      23 bytes total
8-bit   +1      24 bytes total
8-bit   +1      25 bytes total      requires 32 bytes allocated

Total Used  = 25 bytes
Allocated   = 32 bytes (closest multiple of 8-bytes [64-bits])
```

4.4.3.1 分配和取消分配栈空间

可按以下示例所示分配和取消分配栈空间：

分配 32 个字节（必须是 8 字节的倍数）：

```
ADD.U16    A15,A15,#32    ; SP = SP + 32
```

取消分配 32 个字节（必须是 8 字节的倍数）：

```
SUB.U16    A15,A15,#32    ; SP = SP - 32
```

编译器自动分配和取消分配栈空间，并强制对齐到 64 位字边界。

如果需要且栈大小小于 256 字节，还可以使用 ***(A15++#u8imm)*** 寻址模式推送栈上的某些内容，并且还可以分配额外的栈空间。例如：

```
ST.64   *(A15++#32),XD0     ; Push 64-bit XD0 value on stack, then allocate
                            ; 32 bytes on stack (SP = SP + 32)
```

同样，可以使用 ***(A15 -= #n8imm)*** 寻址模式取消栈分配并从栈中弹出内容。例如：

```
LD.64   XD0,*(A15-=#32)     ; De-allocate 32-bytes from stack (SP = SP - 32),
                            ; and pop 64-bit value from stack into XD0
```

如果需要访问栈上距离大于 8192 字节的值，则需要使用寻址指针来访问该值。例如：要访问距离栈顶 8216 字节的 32 位值，请执行以下操作：

```
SUB.U16     A0,A15,#8216    ; A0 = SP - #8216
LD.32    D0,*A0          ; D0 = contents of stack at SP-8216
```

通常，对于大型栈，编译器会分配其中一个 Ax 寻址模式寄存器作为帧指针，并可以使用可用的指针寻址模式来索引到栈中。

上述方法还可用于在需要频繁访问栈上的局部变量或需要对数据使用指针递增/递减运算的情况下初始化栈内的指针。

请注意，无论使用何种寻址模式，任何栈存储器访问都必须与访问的字大小对齐，并且任何非对齐访问都会产生故障。编译器负责对栈空间上的任何数据进行对齐。

备注

1. 如果使用指针来访问栈内容，则假设该值与正确的字访问大小适当对齐。 如果该值损坏，则可能会导致访问故障。
2. CALL 操作会自动将 RPC（返回 PC）值压入栈并将栈指针递增 8，因此始终保持栈对齐。32 位 RPC 存储在 64 位字的低 32 位中。同样，RET 操作会从栈中弹出 RPC 值，并将 SP 递减 8 以保持栈对齐。如果在执行 CALL 或 RET 操作时栈指针未对齐，则会生成故障。

4.4.4 循环寻址指令

C29x CPU 不像 C28x CPU 那样支持循环寻址的原生寻址模式。但是，C29x CPU 架构中存在的功能并行性可确保缺少原生循环寻址模式不会对性能产生影响。

循环寻址由以循环方式修改寻址寄存器的指令执行。这些是 16 位指令，需要 1 个周期来执行。支持的指令包括：

INC.CIRC Ay,Ax：

将 Ay 递增，直至达到限制 (Ax)，然后将该值复位为 0。

```
if (Ay >= Ax)   Ay = 0
else            Ay = Ay + 1
; where  Ay = A0 to A3
; and    Ax = A0 to A14
```

DEC.CIRC Ay,Ax：

将 Ay 递减，直至达到限制 (0)、然后将值复位为 Ax。

```
if (Ay <= 0)    Ay = Ax
else            Ay = Ay - 1
; where  Ay = A0 to A3
; and    Ax = A0 to A14
```

这种类型的寻址模式通常用于实现有限脉冲响应 (FIR)、最小均方根 (LMS) 或卷积滤波器。

使用 C 语言的典型 FIR 滤波算法：

```
sum = 0;
circ_index = save_circ_index;
for(i=0; i < N_taps; i++)
{
    sum += Data[circ_index] * Coef[i];
    circ_index++;
    if( circ_index >= N_taps)
        circ_index = 0;
}
save_circ_index = circ_index;
```

滤波器的主内核可编码如下（7 抽头 FIR 示例）：

```
LD.32           A0,@save_circ_index ; A0 = circ_index
MV              A6,#N-1             ; A6 = filter taps, N = 7
MV              A4,#Data            ; A4 -> Data Array
MV              A5,#Coef            ; A5 -> Coef Array
LD.32           M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M4,M0,M1
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M5,M0,M1
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M6,M0,M1
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M7,M0,M1
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M4,M0,M1
||SADDF         M6,M6,M4
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M5,M0,M1
||SADDF         M7,M7,M5
||LD.32         M0,*(A4+A0)         ; Read Data From Current Index
||LD.32         M1,*A5++            ; Read Coef, Increment Coef Pointer
||INC.CIRC      A0,A6               ; if(A0 >= A6) A0 = 0 else A0 = A0 + 1
SMPYF           M4,M0,M1
||SADDF         M6,M6,M4
ADDF            M7,M7,M5
ADDF            M6,M6,M4
ST.32           @save_circ_index,A0 ; Save current circ index position
ADDF            M7,M7,M6            ; final sum = M7
```

4.4.5 位反向寻址指令

C29x CPU 不像 C28x CPU 那样支持本机位反向寻址模式。但是，C29x CPU 架构中存在的功能并行性可确保缺少原生位反向寻址模式不会对性能产生影响。

位反向寻址由一条以位反向方式修改寻址寄存器的指令执行，通常用于对快速傅里叶变换 (FFT) 类型算法的数据进行重新排序。

表 4-8. 位反向寻址的可视化表示

地址	值	位反向寻址	位反向值
0000	0	0000	0
0001	1	1000	8
0010	2	0100	4
0011	3	1100	12
0100	4	0010	2
0101	5	1010	10
0110	6	0110	6
0111	7	1110	14
1000	8	0001	1
1001	9	1001	9
1010	10	0101	5
...	...	...	...

位反向寻址支持的指令为：

ADD.BITREV Az,Ay,Ax

执行 ADD 运算，但从左到右添加这些位（与从右到左的标准 ADD 不同）。示例如下：

```
; Ax = 0011 1001
; Ay = 0000 1000
; Az = 0011 0101 (after a bit reversed add):
ADD          Az,Ay,Ax    ; Normal Add:        Az = 0100 0001
ADD.BITREV   Az,Ay,Ax    ; Bit Reversed Add:  Az = 0011 0101
```

以下示例说明了如何使用此运算以位反向顺序对数据数组进行反向：

```
BitReversedIndex        = 0;
BitReversedIncrement    = N/2;
for(i=0; i < N; i++)
{
    BitReversedDataArray[BitReversedIndex] = NormalDataArray[i];
    BitReversedIndex = BitReversedAdd(BitReversedIndex+BitReversedIncrement);
}
```

通常，在对数据进行位反向时，数据数组是大小为 2 的倍数（N = 16、32、64、128 等）。

然后，需要将 BitReversedIncrement 设置为数组大小的一半 (N/2)，以位反向顺序递增 1。

前面运算的汇编代码为：

```
MV              A0,#0                       ; A0 = BitReversedIndex = 0
MV              A8,#N/2                     ; A8 = Increment Step = N/2
MV              A4,#NormalDataArray         ; A4 = Stating Address Of
                                            ;      NormalDataArray
MV              A5,#BitReversedDataArray    ; A5 = Stating Address Of
                                            ;      BitReversedDataArray

; Repeat N times:
LD.32           D0,*A4++                    ; Read From NormalDataArray
ST.32           *(A5+A0),D0                 ; Write To BitReversedDataArray
||ADD.BITREV    A0,A0,A8                    ; Increment BitReversedIndex
LD.32           D0,*A4++                    ; Read From NormalDataArray
ST.32           *(A5+A0),D0                 ; Write To BitReversedDataArray
||ADD.BITREV    A0,A0,A8                    ; Increment BitReversedIndex
....
LD.32           D0,*A4++                    ; Read From NormalDataArray
ST.32           *(A5+A0),D0                 ; Write To BitReversedDataArray
||ADD.BITREV    A0,A0,A8                    ; Increment BitReversedIndex
```

章节 5

功能安全和信息安全单元 (SSU)

功能安全和信息安全单元 (SSU) 将功能安全、存储器管理 (MPU) 和信息安全作为一个功能实现。本章简要概述 SSU 模块。有关 SSU 的详细信息，请参阅 *F29H85x 和 F29P58x 实时微控制器技术参考手册*。

5.1 SSU 概述

SSU 充当 CPU、存储器和外设之间的过滤器或防火墙，并在 CPU 尝试访问芯片上的外设和存储器时执行用户保护策略。需要将闪存、ROM、RAM 和外设等所有器件资源分配到访问保护 (AP) 范围。C29x CPU 上运行的任何代码都通过 APx 区域与链接关联。然后，链接与栈关联，而栈又与特定区域关联。

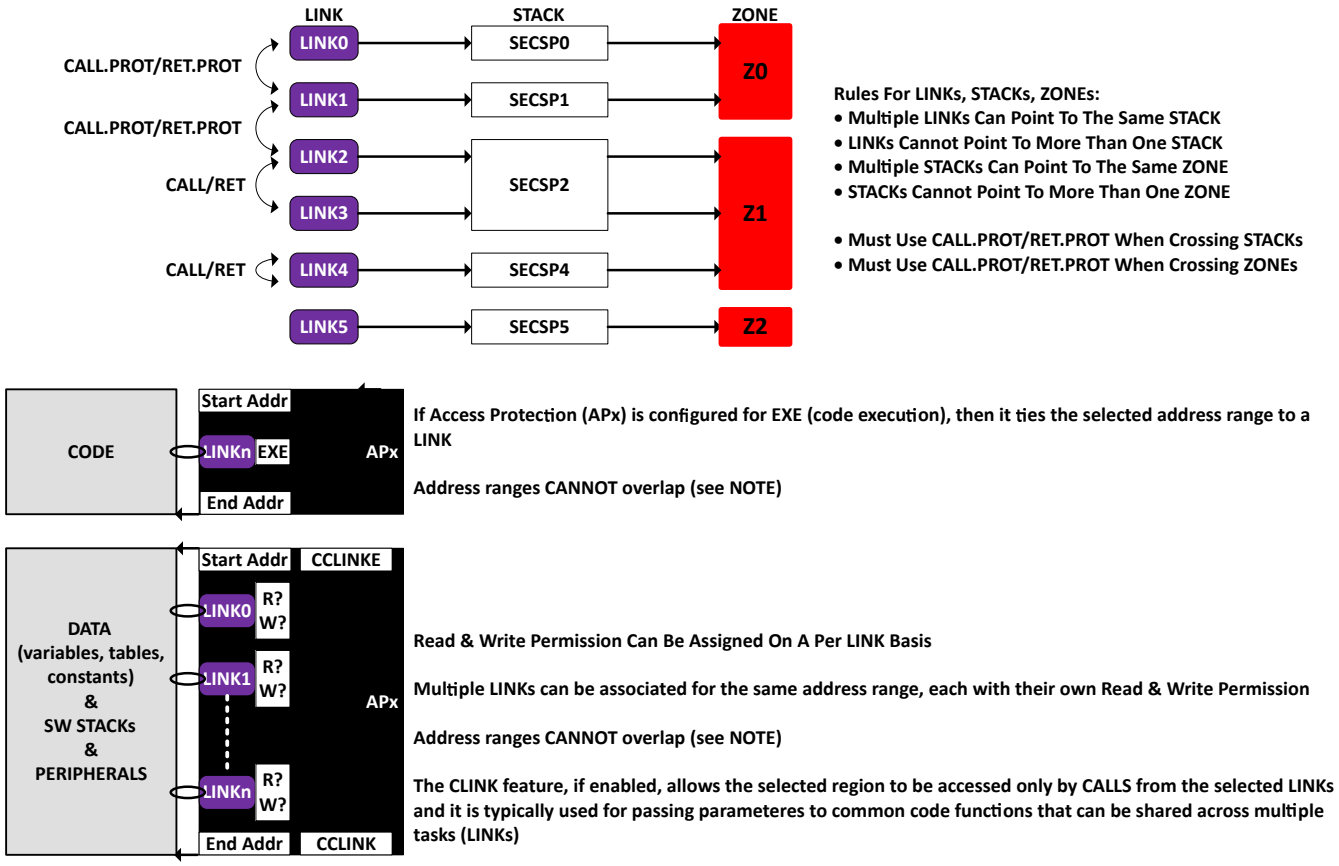

图 5-1. SSU 概述

备注

- 只要操作不在栈之间跳转，就可以使用标准 CALL/RET 操作在链接之间跳转。
- 即使不跨栈，也可以使用 CALL.PROT/RET.PROT，这在最初对代码进行分段后再将每个函数分配到不同的栈时非常有用。
- 适用于 CALL.PROT 的规则也适用于 LB.PROT
- 对于包含多个 C29x CPU 的系统，每个 CPU 都有链接、栈和 AP 区域，而区域在整个器件上共享。

5.2 链接和任务隔离

功能安全和信息安全方法基于任务隔离的概念。本节使用了两个不同任务（一个控制任务和一个通信任务）的示例。任务无法查看或破坏其他任务的唯一程序存储器、数据存储器、软件栈和外设访问。从调试的角度来看，每个安全区域（ZONE1、ZONE2）都有一个安全密码，只能通过使用匹配的密码启用用于调试的区域来访问该密码。每个任务具有一个相关的安全栈指针（SECSP2、SECSP3），当进入相应的任务时，该指针被复制到 CPU 栈指针 SP = A15 中。退出任务时，CPU 栈指针的当前内容被复制到相应的栈指针（SECSP2、SECSP3）中。

C29x CPU 利用链接的概念将执行代码绑定到特定的任务。例如，LINK2 与 SECSP2 和 ZONE1 关联。类似地，LINK3 与 SECSP3 和 ZONE2 相关联，如图 5-2 所示。

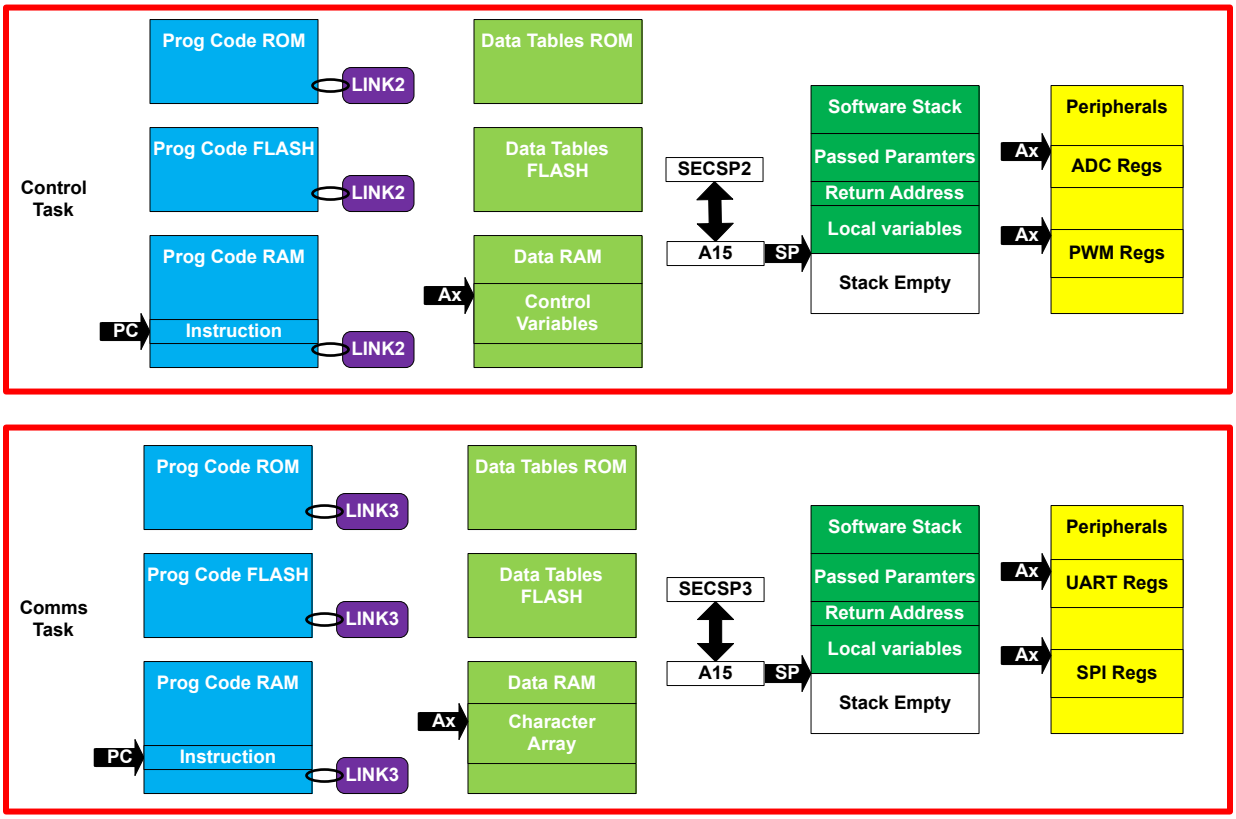

图 5-2. 用于创建任务隔离的链接概念

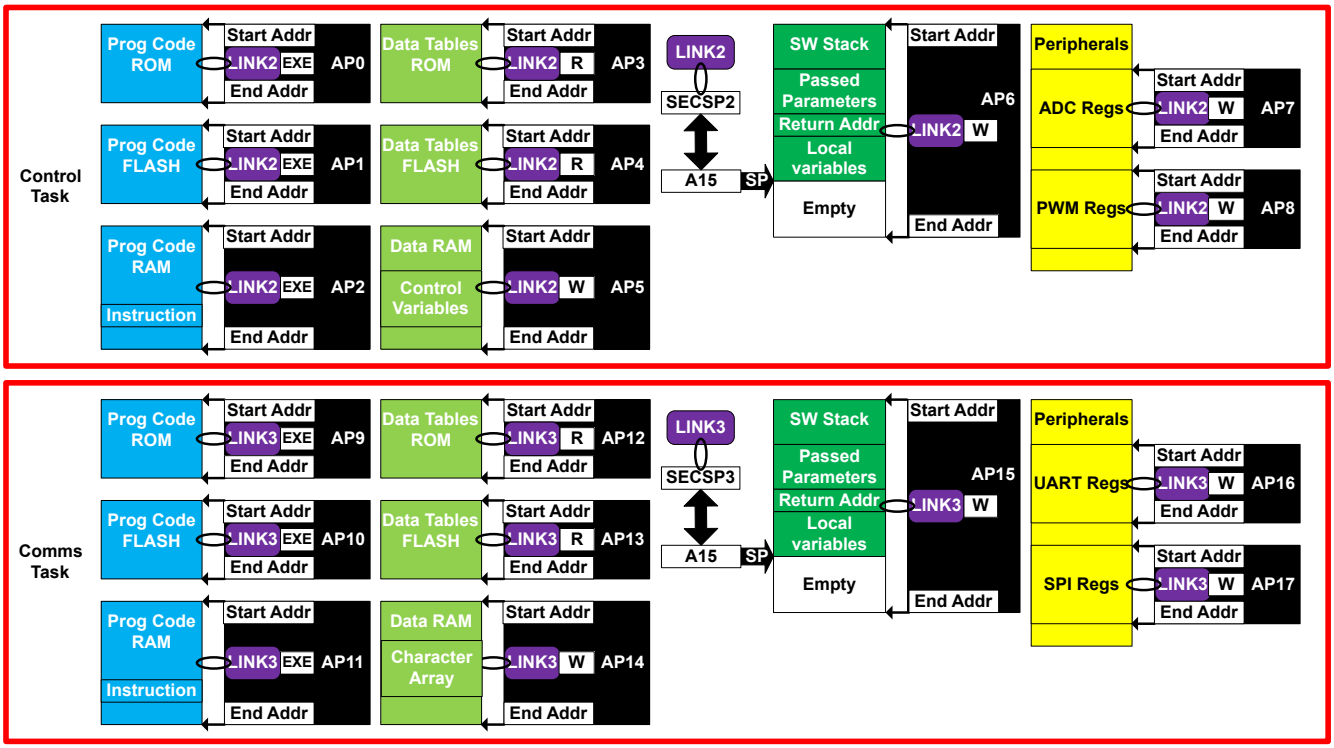

图 5-3. 存储器和外设访问保护的概念

备注

所有访问保护 (AP) 区域的最小地址粒度为 4KB。

5.3 在任务隔离边界之外共享数据

当控制任务 (LINK2) 需要与通信任务 (LINK3) 共享数据时，需要与具有 WRITE 属性的 LINK2 和具有 READ 属性的 LINK3 一起使用共享 RAM（带有 AP18）。同样，当通信任务 (LINK3) 希望与控制任务 (LINK2) 共享数据时，需要与具有 WRITE 属性的 LINK3 和具有 READ 属性的 LINK2 一起使用另一个共享 RAM（带有 AP19）。同样，ADC2 结果寄存器（带有 AP20）分配给具有 READ 属性的 LINK2 和 LINK3。这允许通信和控制任务读取 ADC2 结果寄存器。

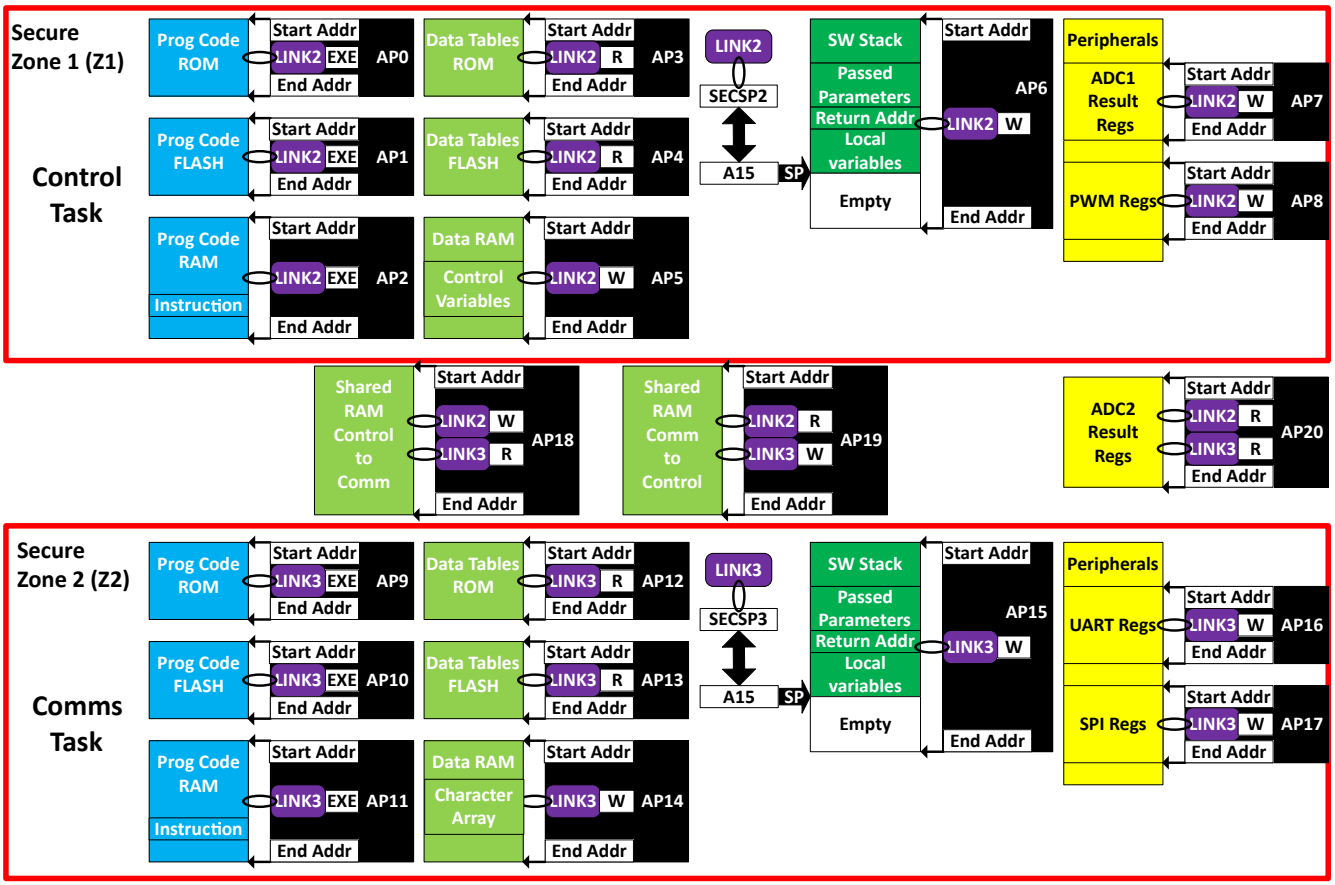

图 5-4. 跨链接共享数据的概念

5.4 受保护的调用和返回

CALL.PROT 操作允许跨栈。到达 CALL.PROT 的目标地址时，ENTRY1.PROT 和 ENTRY2.PROT 指令必须是执行的第一条指令。如果这些指令不存在，CPU 将进入故障状态。从 CALL.PROT 操作返回时，返回地址处的第一条指令必须是 EXIT.PROT 指令。如果 EXIT.PROT 指令不存在，CPU 会进入故障状态。这些进出指令控制着越过安全边界时代码的进入点和退出点，使用户代码能够在使用之前检查任何传递的参数或数据。

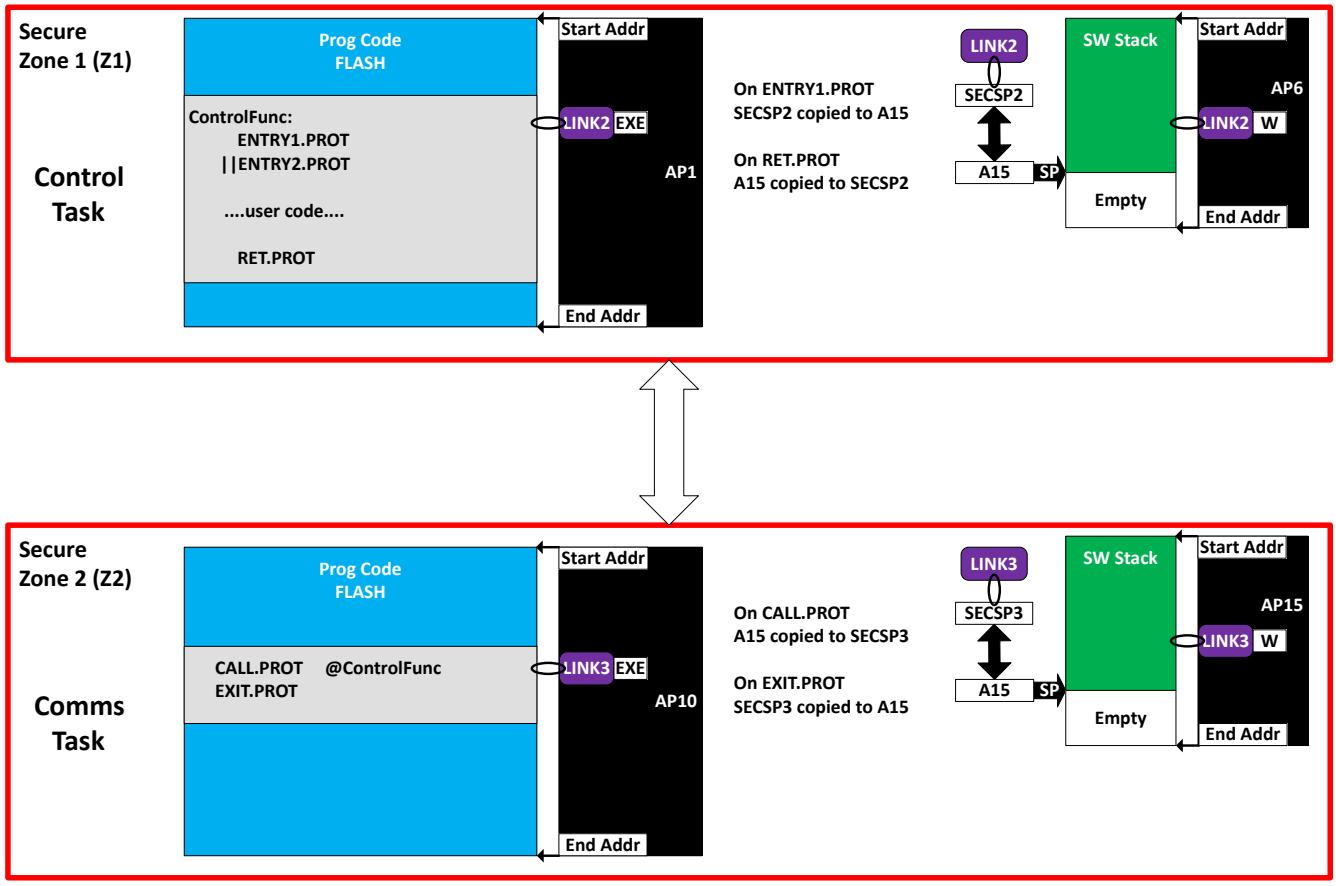

图 5-5. 受保护的调用和返回

章节 6

仿真

CPU 中的调试控制器包含用于高级仿真功能的硬件扩展，可以帮助您开发应用系统（软件和硬件）。本章描述了所有仅使用 JTAG 端口（具有 TI 扩展）的 F29x 器件上可用的仿真特性。

6.1 仿真功能概述

用于高级仿真功能的 CPU 硬件扩展提供了一种简单、经济且与速度无关的 CPU 访问方式，可实现复杂的调试和经济的系统开发，而无需传统仿真器系统所需的昂贵布线和对处理器引脚的访问。

这种访问方式不会侵占系统资源。片上开发接口提供：

- 对内部和外部存储器的极低侵入度
- 对 CPU 和外设寄存器的极低侵入度
- 控制代码执行：
  - 在软件断点指令处中断（指令替换）
  - 在指定的程序或数据访问处中断（无需指令替换）
  - 收到调试主机或其他硬件的外部注意请求时中断
  - 在执行单条指令后中断（单步执行）
  - 控制设备上电后的代码执行
- 以非侵入方式确定器件状态：
  - 检测系统复位、仿真/测试逻辑复位或断电的情况
  - 检测系统时钟或存储器就绪信号是否缺失
  - 确定是否启用全局中断
  - 确定调试访问会被阻止的原因
- 用于性能基准测试的周期计数器。

6.2 调试术语

以下定义有助于理解本章中的信息：

- **调试暂停状态：**器件未执行代码的状态。
- **调试事件：**一种操作，例如软件断点指令解码、出现断点/观察点、外部系统触发器或主机处理器发出的请求，此类操作会导致特殊的调试行为（例如停止器件）。
- **中断事件：**使器件进入调试暂停状态的调试事件。

6.3 调试接口

目标级 TI 调试接口使用 5 个标准 IEEE 1149.1 (JTAG) 信号（TRST、TCK、TMS、TDI 和 TDO）和两个 TI 扩展（EMU0 和 EMU1）。图 6-1 展示了用于将目标连接至扫描控制器的 14 引脚 JTAG 接头，表 6-1 定义了这些引脚。如表 6-1 中所示，该标头需要不止五个 JTAG 信号和 TI 扩展。该接头还需要测试时钟返回信号 (TCK_RET)、目标电源 (VCC) 和接地 (GND)。TCK_RET 是扫描控制器输出并进入目标系统的测试时钟。如果目标系统不提供测试时钟（在这种情况下，无法使用 TCK），则目标系统将使用 TCK_RET。在许多目标系统中，TCK_RET 连接到 TCK 并用作测试时钟。

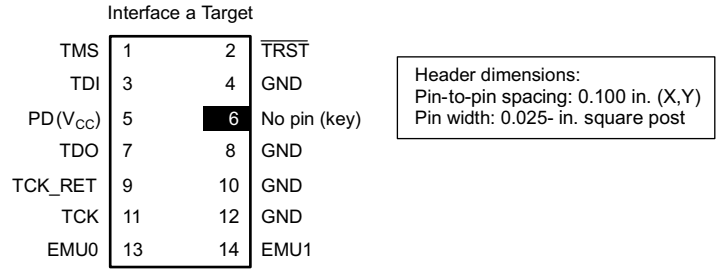

图 6-1. 用以将目标连接到扫描控制器的 JTAG 接头

表 6-1. 14 引脚接头信号说明

信号	说明	仿真器状态 (1)	目标状态 (1)
EMU0	仿真引脚 0	I	I/O
EMU1	仿真引脚 1	I	I/O
GND	接地		
PD (V_{CC})	存在检测。表示仿真电缆已连接并且目标已通电。PD 必须连接至目标系统中的 V_{CC}。	I	O
TCK	测试时钟。TCK 是来自仿真电缆插头的时钟源。该信号可用于驱动系统测试时钟。	O	I
TCK_RET	测试时钟返回。仿真器的测试时钟输入。可以是 TCK 的缓冲或非缓冲版本。	I	O
TDI	测试数据输入	O	I
TDO	测试数据输出	I	O
TMS	测试模式选择	O	I
$\overline{TRST}$ (2)	测试复位	O	I

(1) I = 输入，O = 输出
(2) 不要在 $\overline{TRST}$ 上使用上拉电阻器：器件上有内部下拉电阻器。在低噪声环境中，$\overline{TRST}$ 可保持悬空。在高噪声环境中，可能需要额外的下拉电阻器。（可以根据电流注意事项确定该电阻器的阻值。）

器件加电时 TRST、EMU0 和 EMU1 信号的状态决定了器件的工作模式。一旦器件有足够的功率运行，运行模式就会生效。如果 TRST 信号上升，EMU0 和 EMU1 信号将在上升沿被采样并且锁存工作模式。其中一些模式保留用于测试目的，但在表 6-2 中详细介绍了可在目标系统中使用的模式。目标系统不需要支持正常模式以外的任何模式。

表 6-2. 通过使用 $\overline{TRST}$、EMU0 和 EMU1 来选择器件运行模式

$\overline{TRST}$	EMU1	EMU0	器件运行模式	JTAG 电缆是否激活？
低电平	低电平	低电平	**外设模式**。禁用 C29x CPU 及其存储器部分。另一个处理器将 C29x CPU 视为外设。	否
低电平	低电平	高电平	保留以供测试	否
低电平	高电平	低电平	**复位等待模式**。延长器件的复位时间，直到通过外部方法释放。这使得 C29x CPU 能够在复位中上电，前提是外部硬件仅在上电复位激活时才将 EMU0 保持在低电平。	是
低电平	高电平	高电平	**禁用仿真的正常模式**。这是未连接扫描控制器（如 XDS510）时必须在目标系统上使用的设置。在 C29x CPU 内，TRST 被下拉，EMU1 和 EMU0 被上拉；这是默认模式。	否
高电平	低电平或高电平	低电平或高电平	**启用仿真的正常模式**。这是连接扫描控制器时在目标系统上使用的设置（扫描控制器控制 TRST）。器件上电期间 TRST 一定不能为高电平。	是

6.4 执行控制模式

C29x CPU 支持停止模式调试执行控制模式。STOP 模式可以完全控制程序执行，并允许禁用所有中断。在这种执行模式下，程序在出现中断事件时暂停执行，例如出现软件断点指令或指定的程序空间或数据空间访问。

停止模式会导致中断事件（例如断点和观察点）在下一个中断边界（通常与下一指令边界相同）暂停程序的执行。当执行暂停时，所有中断（包括 NMI 和 RS）都将被忽略，直到 CPU 再次接收到运行代码的指令。

在停止模式下，CPU 可以在以下执行状态下运行：

- **调试暂停状态：**在停止模式调试暂停状态下，CPU 暂停。当 CPU 在启用调试的情况下运行并遇到中断事件（例如断点或观察点、硬件触发器或用户启动的停止请求）时，将进入此状态。
  - 用户暂停：用户从调试器发出调试暂停请求。
  - 硬件断点：可以设置 ERAD 以在指定的程序地址上生成硬件断点。如果指定地址处的指令包即将进入 CPU 流水线的 Decode2 阶段，则会使 CPU 进入暂停状态。
  - 软件断点：这是由调试器通过将 EMUSTOP0 指令放置在所需的程序地址来设置的。如果 EMUSTOP0 即将进入 CPU 流水线的 Decode2 阶段，这会导致 CPU 进入暂停状态。
  - 观察点：ERAD 可配置为在 CPU 进行指定的数据存储器访问或某种其他系统条件或这些条件的组合时生成观察点。一旦这个定义的事件发生，ERAD 就会向调试控制器生成一个观察点请求，该请求会导致 CPU 停止。
  - 外部触发器：来自 CPU 之外各种来源的器件级触发器可配置为向 CPU 发出暂停请求，也可导致 CPU 暂停。当暂停一个 CPU 需要在调试器控制多个 CPU 时触发另一个 CPU 的暂停时，通常使用此方法。

  在调试暂停状态下，由于 CPU 暂停，CPU 无法处理任何中断，包括 NMI 和 RS（复位）。当 CPU 处于暂停调试状态时，如果同一中断的多个实例发生而第一个实例未被处理，则稍后的实例将丢失。
- **单步状态**：当用户指示调试器执行单个指令包时，进入此状态。CPU 执行 PC 指向的单个指令包，然后返回到调试暂停状态（CPU 从一个中断边界执行到下一个中断边界）。只有在单指令完成前，CPU 才处于单指令状态。如果在此状态下发生中断，则用于进入此状态的命令决定是否可以为该中断提供服务。如果 DINT 被置位，CPU 可以处理中断；如果 DINT 未被置位，即使中断是 NMI 或 RS，CPU 也无法处理中断。
- **运行状态**：当用户在启用停止调试模式的情况下从调试器接口发出运行命令时，从停止状态进入该状态。CPU 执行指令，直到调试器命令或调试事件使 CPU 返回到调试暂停状态。CPU 可以处理此状态下的所有中断。当中断与调试事件同时发生时，调试事件具有优先级；但是，如果中断处理在调试事件发生之前开始，则无法处理调试事件，直到中断服务例程开始。
- **自由运行**：当用户在禁用停止模式调试模式后从调试器接口发出运行命令时，从停止状态进入此状态。CPU 将继续执行并忽略诸如断点、观察点和触发器等进一步的调试事件，并像调试器不再连接一样继续执行。
- **同步运行**：这仅仅是基本运行状态的扩展。根据配置，调试控制器可配置为接收运行请求，以便 CPU 仅在某个全局同步信号变为活动状态时启动实际运行。当在由调试器控制的多个 CPU 上同时开始执行时，使用此方法。

6.5 断点、观察点和计数器

6.5.1 软件断点

CPU 支持 ESTOP 指令；当调试器连接后，该指令可用于在 CPU 执行时触发停止。当调试器未连接时，该指令作为 NOP 执行。

6.5.2 硬件调试资源

每个基于 C29x CPU 的系统都有一个 ERAD (嵌入式实时分析和诊断) 模块，可辅助提供调试和系统分析功能。这些功能既可在连接调试器的情况下使用，也可用作实时应用的一部分。两个主要元件是增强型总线比较器块 (EBC) 和系统事件计数器块 (SEC)，后者具有一个可选的 PC 跟踪模块。EBC 和 SEC 的实例数取决于器件。

6.5.2.1 硬件断点

ERAD 增强型总线比较器 (EBC) 模块是由许多相同的总线比较器单元组成的可扩展模块。这些单元可生成硬件断点。硬件断点的作用类似于软件断点指令 (在本例中为 ESTOP0 指令) ，但不需要修改应用软件。硬件断点允许屏蔽地址位。此外，硬件断点允许屏蔽地址位，允许仅使用一个 EBC 资源在地址范围内触发断点。硬件断点会触发调试事件，这会在执行指令之前停止 CPU。总线比较器监视程序地址总线，将内容与参考地址和位掩码值进行比较。

6.5.2.2 硬件观察点

ERAD 增强型总线比较器 (EBC) 模块总线比较器单元，通过监测数据读取地址总线或数据写入地址总线来生成指向 CPU 的硬件观察点。

当某个地址或地址和数据与比较值匹配时，硬件观察点会触发调试事件。地址部分与参考地址和位掩码进行比较，数据部分与参考数据值和位掩码进行比较。

在比较两个地址时，您可以设置两个观察点。比较地址和数据值时，您只能设置一个观察点。执行读取观察点时，该地址比数据早几个周期可用；观察点逻辑会考虑这一点。

执行停止的点取决于观察点是读取还是写入观察点，以及观察点是地址还是地址/数据读取观察点。在以下示例中，访问地址 X 时会发生读取地址观察点，且 CPU 停止，指令计数器 (IC) 指向该点后三条指令。

对于需要地址和数据匹配的读取观察点，CPU 会在 IC 指向该点后六条指令时停止。

在以下示例中，访问地址 Y 时会发生写入地址观察点，且 CPU 在 IC 指向该点后六条指令时停止。

6.5.2.3 基准计数器

系统事件计数器模块由许多相同的计数器单元组成。可用单元的数量取决于具体器件。这些单元可用于各种类型的系统场景，例如：

1. 将计数器用作简单的系统计时器
2. 系统事件（如中断、关键系统事件等）的计数
3. 根据计数器阈值生成中断/事件
4. 配置代码段
5. 测量代码段中的等待状态数
6. 系统事件之间的计数持续时间
7. 计算指定存储器读取和写入之间的持续时间
8. 计算指定存储器读取/写入和系统事件之间的持续时间
9. 测量在一对事件之间所用的最短和最长时间，在多次迭代中进行测量
10. 将计数器链接到链接事件或创建更大的计数器。

可通过调试器和应用软件访问该模块。通过访问应用软件，即使没有调试器，也可以使用调试和分析功能。这在很多实时系统中至关重要，因为并不总是可以连接调试器并执行侵入式调试。在此类情况下，用户代码会设置和控制这些模块，并且仍然能够在不干扰最终应用的情况下调试和分析系统。

6.5.3 PC 跟踪

该 ERAD 模块有一个可选的程序计数器跟踪块，有助于跟踪 PC 不连续/跳跃，进而有助于跟踪在任何给定时间点执行的完整软件序列。该模块只跟踪指令提取/执行中的不连续性（非顺序），可使用软件轻松地重建顺序代码执行。跟踪完成/停止后，可以使用调试器读出跟踪数据以重建代码执行序列。有多种跟踪模式可控制何时启用跟踪以及何时根据 ERAD EBC 事件或某些关键系统级事件生成的事件禁用跟踪。

跟踪数据被存储在一个 RAM 空间中，此空间可由与跟踪有效性相关的附加状态信息一同读取，此信息可被用于重建代码执行序列。对于每次中断，都会存储两个 PC 值，即不连续源和目标。

修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from NOVEMBER 7, 2024 to MARCH 31, 2025 (from Revision * (November 2024) to Revision A (March 2025))	Page
• 删除了特定于设计的信息。	5

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。 严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265