Technical White Paper

C29 CPU：利用 C2000™ MCU 上经过优化的架构提供无可匹敌的实时性能

Saya Goud Langadi, Chen-Yu Chang, Sira Rao

摘要

为了满足实时应用中对更高功率密度和复杂控制技术的新兴设计需求，工程师需要具有更大闪存、更强计算能力和更高集成度的高性能 MCU。这些要求可以通过 CPU 架构的创新来满足，例如 TI C2000™ MCU 中的 C29 CPU，它采用 64 位架构，并集成了高级网络安全元件，如功能安全和信息安全单元 (SSU)。SSU 支持在同一 CPU 内运行的线程之间实现上下文隔离，提供运行时信息安全和无干扰 (FFI)，这一功能通常只出现在微处理器中。C29 核心基于 TI 出色的 C28 核心，针对通用和数字信号处理应用提供更高的性能。

本白皮书讨论了 C29 核心架构和 SSU 的优势，并描述了包含 C29 核心的 MCU 与其他 CPU 架构的 MCU 在多个性能基准测试中的对比结果。本白皮书还介绍了 C29 并行架构的优势，以及使用 C29 编译器实现的性能提升。

内容

插图清单

表格清单

商标

1 实时控制简介

在实时控制中，闭环系统会收集数据，在控制环路中处理数据，并在指定的时间窗口内进行更新。信号链性能可以量化实时控制性能，性能越高，闭环系统运行速度越快。实时控制系统通常由三个主要元素组成：

- **采样或反馈采集**：该应用需要以精确的方式并在非常精确的时刻测量多个关键参数 (电压、电流、电机速度、电机位置和温度) 。
- **处理和控制：**使用采样信息将控制算法应用于输入数据并计算下一个输出命令。
- **驱动**：将计算得出的输出命令应用于系统，从而控制输出。改变驱动电力电子系统的脉宽调制器 (PWM) 单元的占空比就是一个驱动示例。

在实时控制中，系统的性能不仅取决于 CPU 的处理能力，还取决于外设访问速度和中断响应速度。这些因素共同构成了实时信号链的概念。

图 1-1 展示了电机控制和数字电源系统典型实时信号链中涉及的各种元件。更好的信号链性能可以在电机控制应用中实现更高的直流总线使用率，并扩展电机的运行速度范围。在数字电源应用中，更好的信号链性能可实现更高的控制环路频率，从而缩小元件尺寸、降低成本。

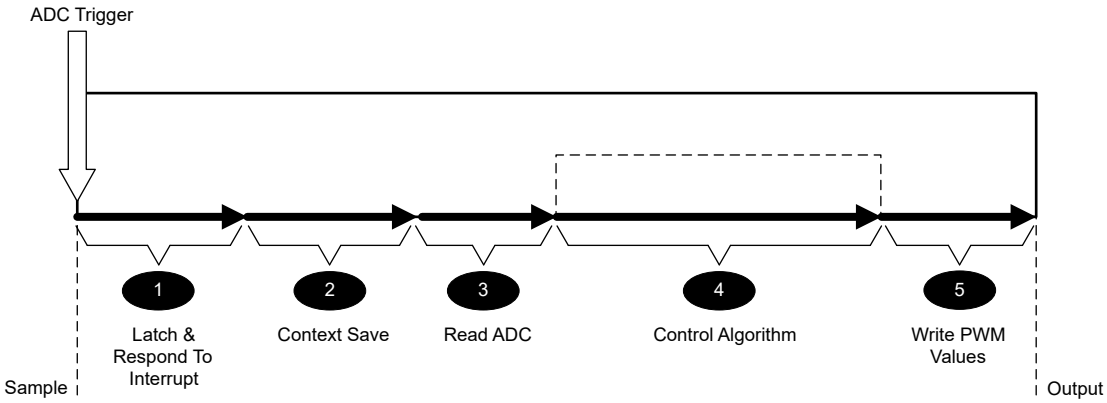


图 1-1. 实时信号链组成

元件 1、2 和 4 由 CPU 架构控制，元件 3 和 5 取决于 CPU 和器件架构。本文件主要重点介绍了元件 1、2 和 4 的改进功能。此外，C2000 MCU 还提供低延时互连，可实现单周期 ADC 读取和单周期 PWM 更新。

2 C29 CPU 及其主要特性

图 2-1 所示为 C29 CPU 的方框图、其主要特性和优势。

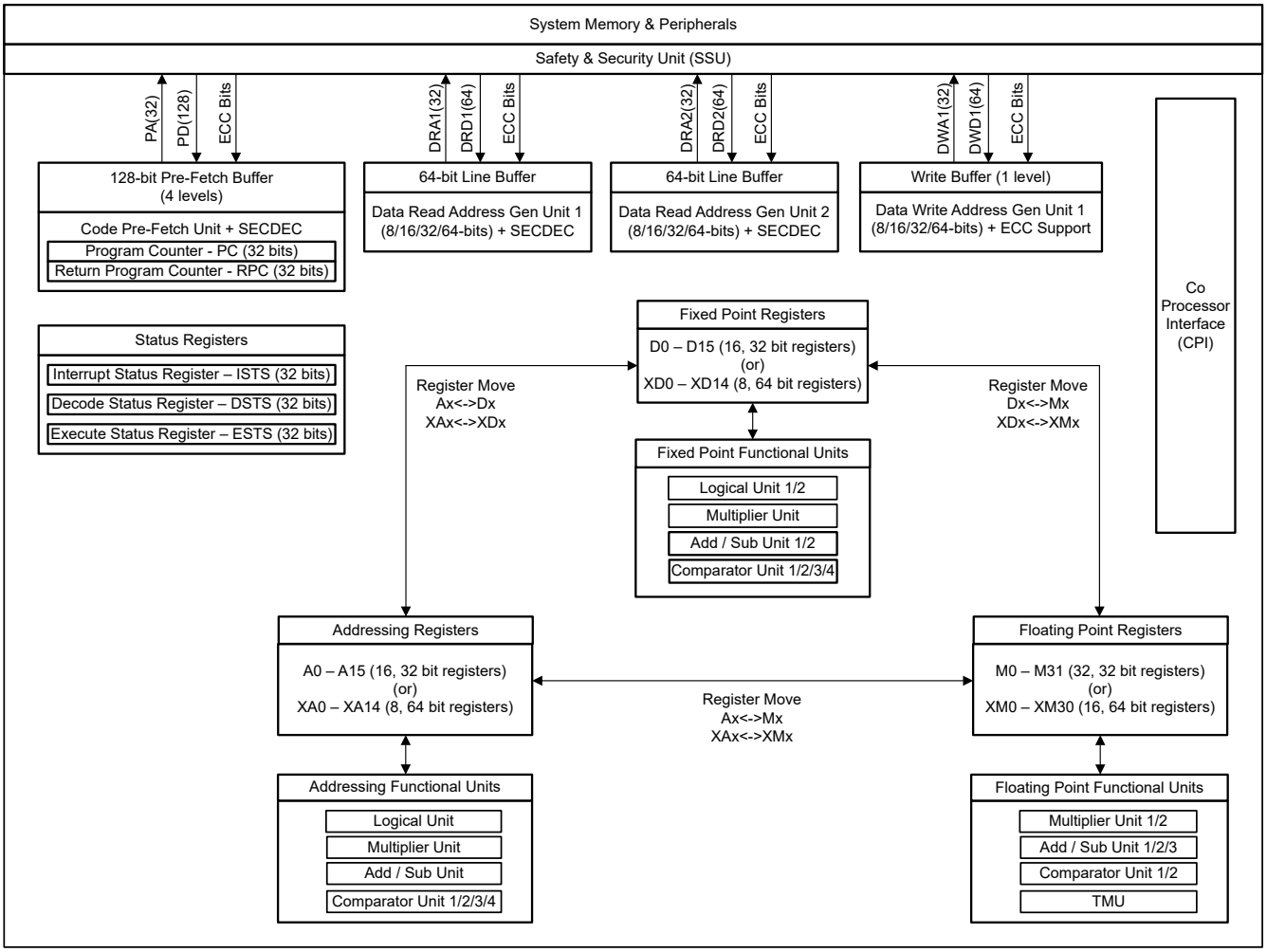


图 2-1. C29 架构方框图

VLIW CPU：C29 基于超长指令字 (VLIW) 架构设计。支持 16 位、32 位和 48 位的可变长度指令。指令数据包大小从 16 位到 128 位，可实现更高的代码密度，同时在单个 CPU 周期内最多执行八条 16 位指令。

CPU 内存总线：128 位宽程序总线，可以提取一个 128 位宽指令数据包以供 CPU 执行。两条 64 位读取总线，支持并行读取两个 64 位数据，同时一条 64 位写入总线可在单个周期内写入 64 位数据到内存。

字节寻址能力和数据类型：C29 支持字节寻址，并且数据类型与其他常见的 CPU 架构（如 ARM）完全兼容。

CPU 寄存器：提供三组寄存器：Ax、Dx 和 Mx。Ax 寄存器包括 16 个 32 位寄存器 (A0-A15) 或 8 个 64 位寄存器 (XA0-XA14)，主要用于地址生成，并在流水线早期阶段执行某些整数运算以提升性能。DX 寄存器包括 16 个 32 位寄存器 (D0-D15) 或 8 个 64 位寄存器 (XD0-XD14)，用于整数定点运算。MX 寄存器包括 32 个 32 位寄存器 (M0-M31) 或 16 个 64 位寄存器 (XM0-XM30)，用于浮点运算。

功能单元：总共有 24 个功能单元，分别与 Ax、Dx 和 Mx 寄存器组及特殊功能寄存器相关联。每个功能单元支持一组特定指令，其中某些功能单元存在多个实例。例如，AX 寄存器文件配有 4 个比较单元，每个周期可以评估两条 switch 语句的情况，从而提高 switch 语句的执行效率。MX 寄存器组配有 2 个浮点乘法单元和 3 个浮点加法/减法单元，能够每两周期执行一次 FFT 蝶形变换运算。

三角函数加速器 (TMU)：支持三角运算，扩展了对 64 位双精度浮点运算的支持，同时兼容 32 位单精度浮点运算。

中断：C29 支持常规中断 (INT) 和优化型实时中断 (RTINT)。RTINT 使用专用的硬件中断堆栈，在 RTINT 发生时，CPU 上下文会自动保存到此堆栈中。这种方式比基于软件的上下文保存机制要快。除了速度更快之外，周期数也是固定的，因此可以提高确定性。而基于软件的上下文保存机制可能需要可变数量的周期数。支持硬件中断优先级排序，减少了通过软件确定优先级的开销。

功能安全：如需实现较高的 ASIL 等级要求，需要在单个或多个 CPU 内运行的多线程代码之间实现隔离。功能安全和信息安全单元 (SSU) 提供了这种隔离能力。SSU 虽然形式简单，但是允许用户定义多个关联的存储器区域（称为访问保护区域 (APR)），这些区域可以通过一个称为 LINK 的概念关联在一起，形成一个隔离的线程。一个线程由代码、数据、堆栈和外设组成。特定代码 LINK 可以通过读取、写入或两者兼具的权限访问特定数据 LINK。相对于传统 MPU，SSU 的优势权限是基于正在执行的代码强制实施的。因此，不需要重新编程 MPU。每个线程都有一个硬件堆栈，堆栈在 CPU 中自动切换，实现完全隔离。在操作系统（例如 AUTOSAR）中，这种高效切换使实时 ISR 可作为 CAT1 中断独立运行，不受操作系统干扰，并与 AUTOSAR 应用程序完全隔离。因此，单个 C29 CPU 内核可以运行操作系统和控制任务，而不会影响控制性能。

信息安全：当代码执行在不同线程的堆栈之间切换时，强制执行入口点和出口点。入口点和出口点指预先定义的、一个线程调用或分支到另一个线程，或从另一个线程返回的点。如果调用、分支到其他地址，或者从其他地址返回，则会触发异常，从而避免安全攻击。SSU 还支持通过称为 ZONE 的机制进行固件更新和调试。每个 ZONE 都具有独立的密码和调试设置。ZONE 支持安全的多方写作开发，每个开发方都可以独立定义密码，阻止其他方查看代码，同时控制代码的调试权限。

表 2-1. C29 主要特性

特性	备注
易于使用	• 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 • 在没有缓存存储器的情况下进行确定性执行
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 • 针对决策代码和实时控制的专门指令（例如：if-then-else 语句，三角和多相向量转换操作）
提高总线吞吐量	• 每个周期能够获取多达 128 位指令字 • 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
代码效率	• 支持可变长度指令集（16 位、32 位和 48 位） • 关键操作编码为 16 位和 32 位操作码，以提高代码密度 • 丰富的指令集通过最简洁的指令优化了运算
ASIL-D 安全能力	• 支持锁步和分离锁定模式 • 集成 ECC 逻辑可实现端到端安全互连 • 使用 SSU 可以完全隔离单独的代码线程（包括堆叠） • 在硬件中从一个线程切换到另一个线程的零 CPU 开销自动实现了出色的实时性能
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 并行架构和编译器优化

C29 ISA 针对特定性能特性设计了一系列指令，旨在优化以下领域的性能：

实时控制和通用处理

MINMAXF：MINMAXF 指令将 M 寄存器中的浮点值限定为其他两个 M 寄存器中指定的下限和上限。

QUADF：QUADF 指令设置 TDM 寄存器（CPU 状态寄存器）的标志位，将二维矢量系统分为 16 个段。通过使用输入坐标值的缩放值，TDM 标志位以六段空间矢量生成方法标识该段。这种方法也可扩展到其他空间矢量变体。

更大限度地减少决策制定代码的不连续性

XC：XC 条件执行指令检查 DSTS 寄存器中的状态标志（如 A.Z、A.N、A.ZV 和 A.Z），并根据选定指令判断是否执行指令。基于标志值，一组指令数据包要么执行指令，要么用作 NOP（无操作）跳过。

SELECT：SELECT 指令根据测试条件，在两个源寄存器（例如 A、D 或 M 寄存器）中进行选择，然后将其内容复制到相同类型的目标寄存器。

特殊分支

CPU 支持“延迟分支”概念，可在实现不连续性的同时保持零开销，详见下文。

支持使用测试标志（如 TA.MAP 和 TDM.MAP）、基于 LUT 函数的条件分支指令。

QDECB：QDECB 多路条件分支指令检查 A14 寄存器的内容。基于 A14 寄存器的值，程序执行要么分支到四个指定目标中的一个，要么继续执行下一个指令数据包，并对 A14 寄存器值进行递减。

DDECB：DDECB 多路条件分支指令检查 A14 寄存器的内容。基于 A14 寄存器的值，程序执行要么分支到两个指定目标中的一个，要么继续执行下一个指令数据包，并对 A14 寄存器值进行递减。

C29 高级编译器根据需要自动选择合适的指令。在某些情况下，例如使用 QUADF 指令时，编译器可通过 C 代码中的内置函数调用相应指令，从而获得优化性能。

指令集用户指南列出了每条指令对应的内置函数（如有）。

---

3 C29 性能基准测试

C29 CPU 的设计目标是在相同工作频率下提供至少是 C28 CPU 2 倍的性能。本节介绍 C29 与 C28 以及与竞品 CPU 之间的性能基准测试结果，并分别针对其架构和编译器的优势进行分析，提供洞见。

除非特别说明，测试均在编译器速度优化设置（C29 编译器采用 -O3）下进行，在零等待状态的存储器中运行。C29 的基准性能结果可能会随着编译器的更新而提升，目前的测试结果基于 0.1.0.STS 版本的编译器。

同样，对于竞品器件，基准测试也是在编译器速度优化设置下进行，在零等待状态的存储器中运行。

3.1 使用 ACI 电机控制的信号链基准测试

ACI 电机控制基准测试模拟无传感器交流感应电机控制应用程序。该应用程序执行所有典型操作：模数转换器 (ADC) 读取相电流信号、转换模块处理这些电流信号、PWM 写入以控制相电压信号。无需特殊的外部硬件即可提供激励，因为该应用程序中的代码块可对感应电机的行为进行建模。为了模拟闭环行为，电机模型的预期电流通过 DAC 模块馈入 ADC 中。单个 ADC 配置为通过两个通道按顺序感应 A 相电流和 B 相电流。C 相电流由 A 相和 B 相电流计算得出，无需直接感测。三个 PWM 写入操作模拟控制三相 A、B 和 C 电压的占空比。

图 3-1 表示了基准测试应用程序的控制环路中断例程中的执行块。控制环路中断以 2KHz 的速率触发，并在应用程序终止之前执行 1024 次控制环路中断例程的迭代。“ACI 模型”和“反向 Clarke 变换和 DAC 输出”块表示代码块，用于在基准测试中模拟电机行为，并非真实的 ACI 电机控制应用的一部分。

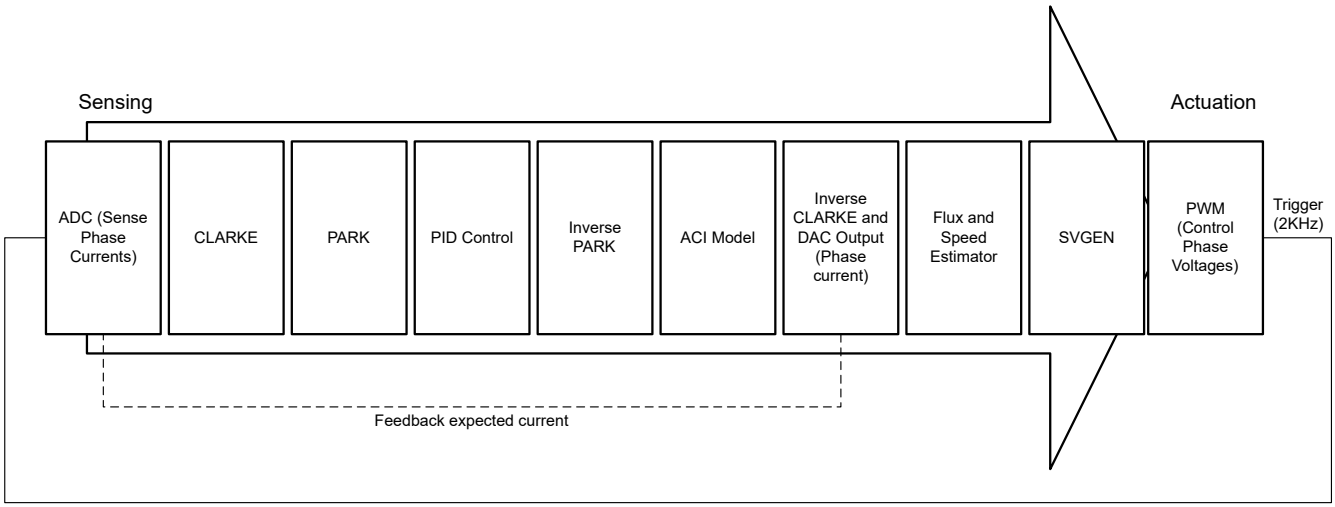


图 3-1. 实时控制环路

实时控制 MCU 的信号链性能 总结了针对实时控制应用的各种竞品 MCU 的实时信号链性能。结果包括几个显著点：

- 与 C28 和同类竞品 MCU 相比，配备 C29 CPU 的 F29H85x 在运行信号链基准测试时所需的 CPU 周期最少。

- 具有 C29 CPU 的 F29H85x 比具有 Cortex-M7 CPU 的竞品 MCU (*1*) 快 4.31 倍（以周期计算）
- 尽管 F29H85x 的运行频率为 200MHz，以 480MHz 的竞品 MCU *1* 为基线，F29H85x 每个 CPU 内核的有效速度（eMHz/内核）达到了 862MHz (4.31 x 200)。竞品 MCU *1* 需要在 862MHz 下运行，才能匹配 F29H85x 在 200MHz 下的信号链性能。

表 3-1. 实时控制 MCU 的信号链性能

MCU	CPU	CPU 类型	CPU 频率	加速器	周期	性能比率	eMHz/内核
1	Cortex-M7	6 级超标量流水线、分支预测	480	–	1094	1	480
2	Cortex-M4	3 级流水线、分支预测	170	CORDIC	838	1.30	220
3	专有 A	4 级超标量流水线（双发射）、分支预测	300	–	857	1.28	384
4	专有 B	5 级流水线、有限双发射	200	TFU	894	1.22	244
5	专有 C	5 级流水线	240	–	1295	0.84	202
AM263P	Cortex-R5F	8 级流水线、有限双发射、分支预测	400	TMU	705	1.55	620
F2837x	C28	8 级流水线、有限双发射	200	TMU	527	2.08	416
F29H85x	C29	9 级流水线 VLIW（支持多达 8 条指令）	200	TMU	254	4.31	862

3.2 实时控制和 DSP 性能

C29 CPU 在实时控制和 DSP 操作方面非常高效，以下基准测试充分证明了其能力：

- CFFT - 复数快速傅里叶变换
- FIR - 有限脉冲响应滤波器
- IIR_sample - 无限脉冲响应滤波器的单个输入样本
- IIR_loop - 无限脉冲响应滤波器的输入样本块
- DCL - 数字控制库（包含 PI、PID 等）
- FCL - 快速电流环路
- SPLL - 软件锁相环
- SVGEN - 空间矢量生成
- FOC - 电机控制中的磁场定向控制（与 ACI 信号链性能测试相同）
- Bin_LUT - 二进制 LUT 搜索

图 3-2 展示了 C29 与 C28 在上述基准测试中的性能对比。在所示基准测试中，C29 的性能 *（以周期数计）* 平均比 C28 高 3 倍。

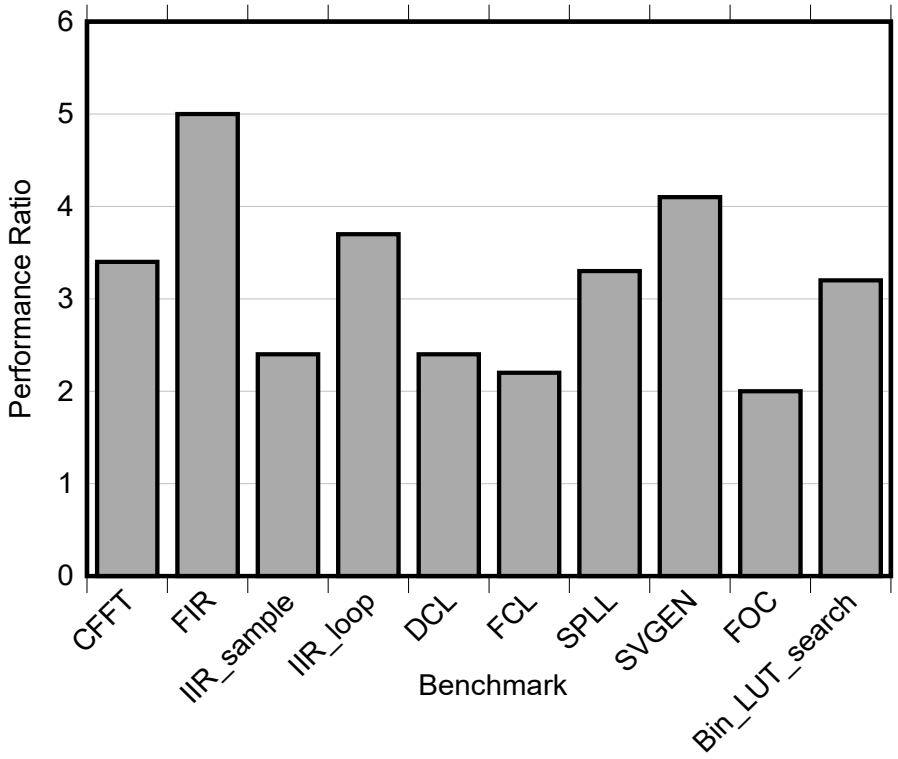


图 3-2. C29 与 C28 的实时控制和 DSP 性能对比

图 3-3 展示了 C29 与 Cortex-M7 在上述基准测试中的性能对比。在所示基准测试中，C29 的性能（**以周期数计**）平均比 Cortex-M7 高近 4 倍。

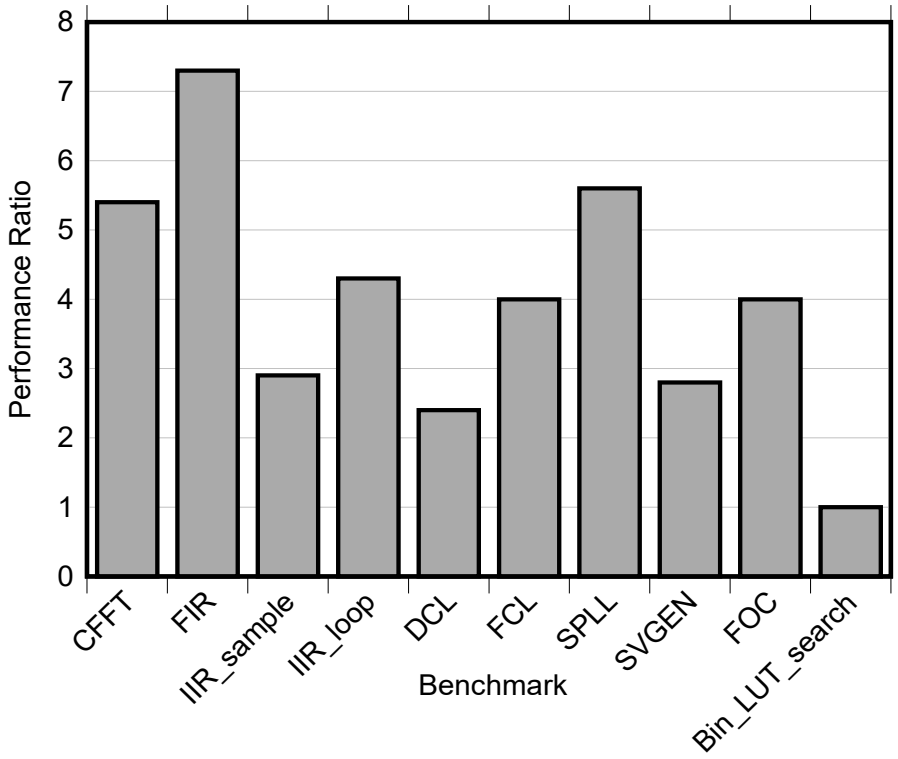


图 3-3. C29 与 M7 的实时控制和 DSP 性能对比

图 3-4 展示了 C29 与某专有 CPU 在上述基准测试中的性能对比。在所示基准测试中，C29 的性能（*以周期数计*）平均比某热门专有 CPU 高近 4 倍。

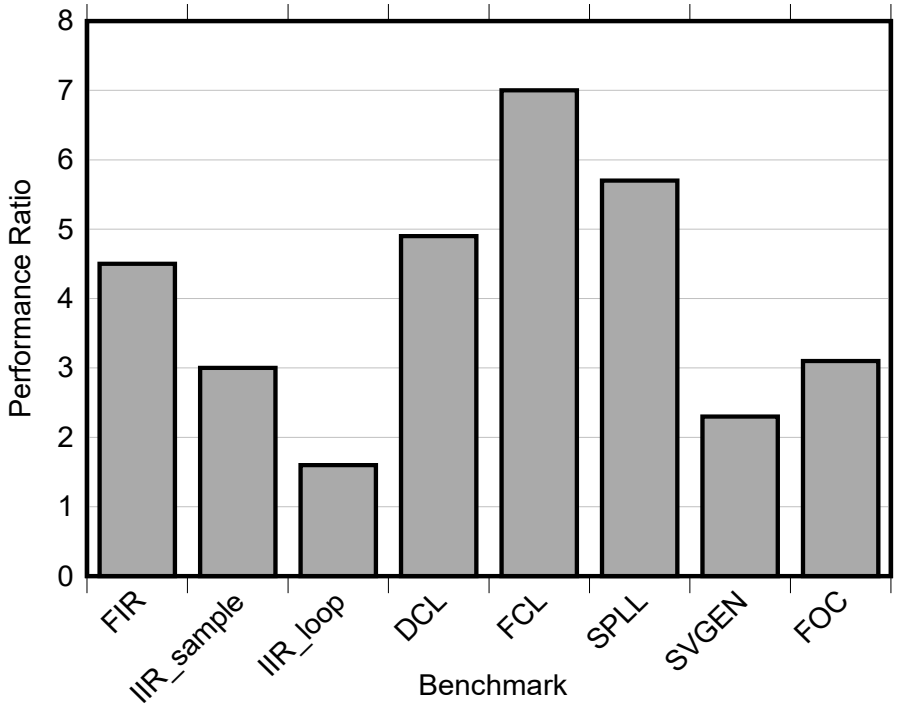


图 3-4. C29 与某专有 CPU 的实时控制和 DSP 性能对比

3.2.1 影响结果的示例和因素

本节提供对于架构和编译器的见解和分析，以帮助理解上面所示的结果。

3.2.1.1 饱和（或限制）示例

在实时应用中，饱和类型代码十分常见。以下展示了在 C 语言中通过两种不同方式实现饱和的总结。

下面的代码块显示了基于 if..else 实现饱和的方法。C29 需要 11 个周期且不依赖输入，性能优于 Cortex-M7（需要 14-27 个周期，且依赖输入）。在 C29 上，if () 是通过条件分支指令 (BC) 实现的，而对于剩余的两条路径（elseif 和 else），则依次使用了比较指令 (CMPF) 和条件指令 (XCP)，从而避免了分支。

```
volatile float in;
volatile float out;
const float max =1.0f;
const float min = -1.0f;
if(in > max)
 out = max;
else if(in < min)
 out = min;
else
 out = in;

C29 Implementation
LD.32   M1,@in
||ONEF M0
CMPF    TDM0,M.GT,M1,M0
ONEF    M1
|| BC    @($LBB0_2),TDM0.NZ
LD.32   M1,@in
|| NEGONEF M2
CMPF    TDM0,M.LT,M1,M2
XCP     #0x1,TDM0.Z
|| LD.32 M1,@in
SELECT TDM0,M1,M2,M1
$LBB0_2:
ST.32 @out,M1
```

```
M7 Implementation
MOVW R0,#in2
MOVT R0,#in2
MOVS R1,#+1
MOVT R1,#+16256
VLDR S0,[R0, #0]
VMOV S1,R1
VCMP.F32 S0,S1
FMSTAT
BLT.N saturation_0
MOV R2,#+1065353216
STR R2,[R0, #+4]
B   saturation_2
saturation_0:
VMOV.F32 S0,#-1.0
VLDR  S1,[R0, #0]
VCMP.F32 S1,S0
FMSTAT
BPL.N saturation1_1
VSTR S0,[R0, #+4]
B  saturation_2
saturation_1:
LDR R1,[R0, #+0]
STR R1,[R0, #+4]
saturation_2:
```

下面的代码块显示了基于三元运算符 '?' 实现饱和的方法。C29 通过 MINMAXF 指令实现，且无需分支，需要 3 个周期且不依赖输入，性能优于 Cortex-M7（需要 18-22 个周期，且依赖输入）。

```
volatile float in;
volatile float out;
const float max =1.0f;
const float min = -1.0f;

float temp = in;
temp = (temp > max)? max: ((temp < min)? min: temp);
out = temp;

C29 Implementation
ONEF    M0  || LD.32 M1,@in  || NEGONEF M2 MINMAXF M1,M0,M2 ST.32   @out,M1

M7 Implementation
MOVW R0,#in2
MOVS R1,#+1
MOVT R0,#in2
MOVT R1,#+16256
VMOV S2,R1
VLDR S0,[R0, #0]
VCMP.F32 S0,S2
VMOV.F32 S1,S0
FMSTAT
IT  GE
VMOVGE.F32 S1,#1.0
BGE.N saturation_0
VMOV.F32  S2,#-1.0
VCMP.F32  S0,S2
FMSTAT
IT  MI
VMOVMI.F32 S1,S2
saturation_0:
VSTR  S1,[R0, #+8]
```

C29 编译器经过优化，能够在 if..else 或三元运算符实现方式之间，生成相同性能的代码。

|| 表示与上述指令并行执行的指令。

3.2.1.2 死区示例

在实时应用中，死区代码经常发生。

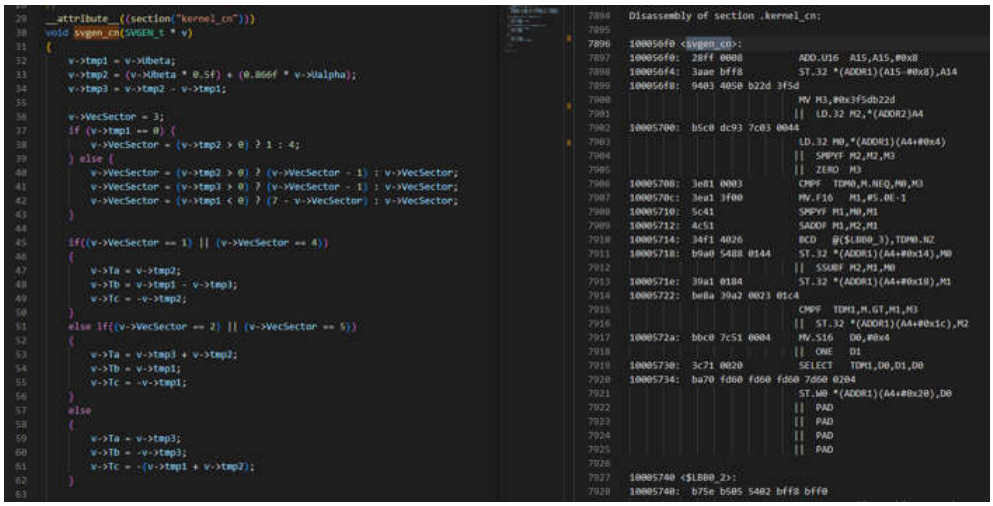
下面的代码块显示了基于三元运算符 '?' 实现死区代码的方法。C29 的性能（需要 10 个周期且不依赖输入）优于 C28（需要 25-36 个周期，且依赖输入），也优于 Cortex-M7（需要 23-35 个周期，且依赖输入）。这种高效性得益于延迟返回指令 (RETD) 和延迟时隙中高效使用的比较 (CMPF) 指令和赋值 (SELECT) 指令。

```
float deadzone(float in)
{
 float out;
 float  out_pos = in - 1.0f;
 float  out_neg = in + 1.0f;
 out = (in > 1.0f)?  out_pos : ((in > -1.0f)? 0.0f : out_neg);
 return out;
}
C29 Implementation
Function call:
CALL @deadzone
|| LD.32 M0,@in1
;---------CALLD occurs
ST.32 @out1,M0
 deadzone:
ONEF M1
|| NEGONEF M2
SADDF M3,M0,M2
|| CMPF TDM0,M.GT,M0,M2
|| SADDF M2,M0,M1
|| RETD
ZERO M4
CMPF TDM1,M.GT,M0,M1
```

```
|| SELECT TDM0,M0,M4,M2
SELECT TDM1,M0,M3,M0

C28 Implementation
Function call: MOVW  DP,#_in1
MOV32 R0H,@_in1
LCR   #_deadzone
MOVW  DP,#_out1
MOV32 @_out1,R0H

_deadzone:
ADDB   SP,#2
CMPF32 R0H,#16256
MOVST0 ZF, NF
B      $C$L1,LEQ
ADDF32 R0H,R0H,#49024
B      $C$L3,UNC
$C$L1:
CMPF32 R0H,#49024
MOVST0 ZF, NF
B      $C$L2,LEQ
ZERO   R0H
B      $C$L3,UNC
$C$L2:
ADDF32 R0H,R0H,#16256
$C$L3:
SUBB   SP,#2
LRETR

M7 Implementation
Function call:
VLDR  S0,[R6, #+144]
BL    deadzone
VSTR  S0,[R6, #+152]

deadzone:
MOVS     R0,#+1
MOVT     R0,#+16256
VMOV     S2,R0
VMOV.F32 S1,S0
VCMP.F32 S1,S2
FMSTAT
VMOV.F32 S0,#1.0
VADD.F32 S0,S1,S0
BLT.N    deadzone_0
VMOV.F32 S3,#-1.0
VADD.F32 S0,S1,S3
BX       LR
deadzone_0:
MVN      R1,#+1082130432
VMOV     S4,R1
VCMP.F32 S1,S4
FMSTAT
ITT     GE
MOVGE   R0,#+0
VMOVGE  S0,R0
BX      LR
```

3.2.1.3 空间矢量生成 (SVGEN) 示例

空间矢量生成 (SVGEN) 是电机控制系统中的常用功能，主要将矢量 (α，β) 映射到 6 段空间矢量，从而生成 3 个 PWM 信号。在普通实现中 (如 图 3-5 中所示) ，使用了 if..else 语句 (图左侧) ，编译器生成包含分支的代码 (图右侧) 。



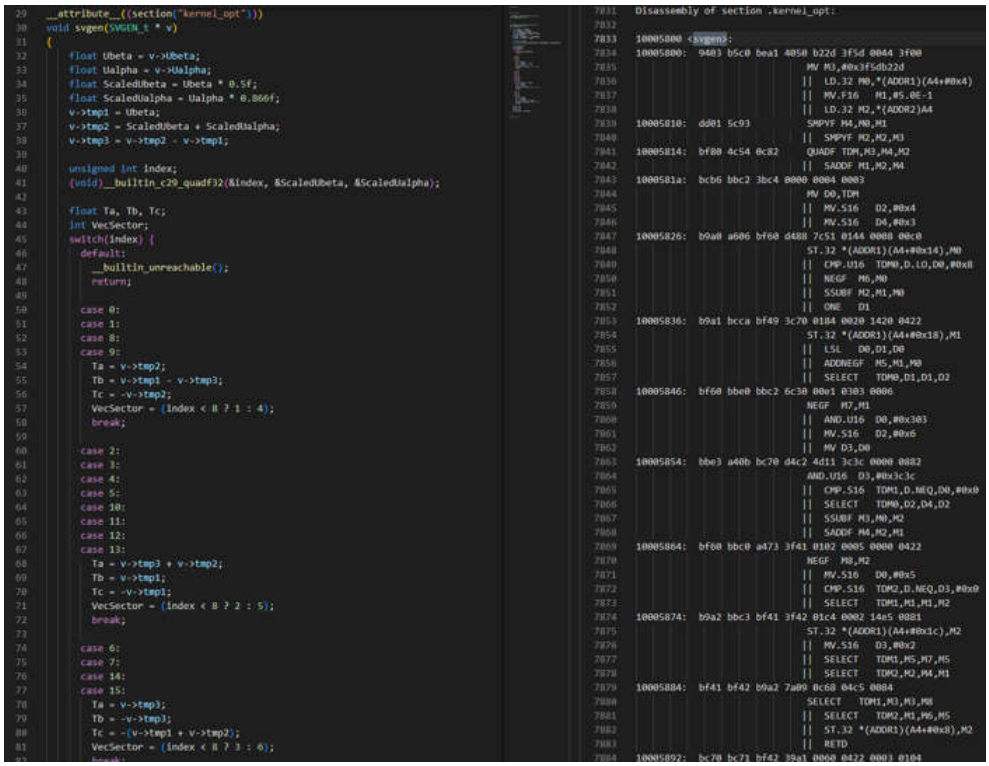
```
__attribute__((section("kernel_cn")))
void svgen_cn(SVGEN_t * v)
{
    v->tmp1 = v->Ubeta;
    v->tmp2 = (v->Ubeta * 0.5f) + (0.866f * v->Ualpha);
    v->tmp3 = v->tmp2 - v->tmp1;

    v->VecSector = 3;
    if (v->tmp1 == 0) {
        v->VecSector = (v->tmp2 > 0) ? 1 : 4;
    } else {
        v->VecSector = (v->tmp2 > 0) ? (v->VecSector - 1) : v->VecSector;
        v->VecSector = (v->tmp3 > 0) ? (v->VecSector - 1) : v->VecSector;
        v->VecSector = (v->tmp1 < 0) ? (7 - v->VecSector) : v->VecSector;
    }

    if((v->VecSector == 1) || (v->VecSector == 4))
    {
        v->Ta = v->tmp2;
        v->Tb = v->tmp1 - v->tmp3;
        v->Tc = -v->tmp2;
    }
    else if((v->VecSector == 2) || (v->VecSector == 5))
    {
        v->Ta = v->tmp3 + v->tmp2;
        v->Tb = v->tmp1;
        v->Tc = -v->tmp1;
    }
    else
    {
        v->Ta = v->tmp3;
        v->Tb = -v->tmp3;
        v->Tc = -(v->tmp1 + v->tmp2);
    }
```

```
Disassembly of section .kernel_cn:

100056f0 <svgen_cn>:
100056f0:  28ff 0008          ADD.U16  A15,A15,#0x8
100056f4:  3aae bff8          ST.32 *(ADDR1)(A15-#0x8),A14
100056f8:  9403 4050 b22d 3f5d
                              MV M3,#0x3f5db22d
                           || LD.32 M2,*(ADDR2)A4
10005700:  b5c0 dc93 7c03 0044
                              LD.32 M0,*(ADDR1)(A4+#0x4)
                           || SMPYF M2,M2,M3
                           || ZERO  M3
10005708:  3e81 0003          CMPF  TDM0,M.NEQ,M0,M3
1000570c:  3ea1 3f00          MV.F16   M1,#5.0E-1
10005710:  5c41               SMPYF M1,M0,M1
10005712:  4c51               SADDF M1,M2,M1
10005714:  34f1 4026          BCD   @($LBB0_3),TDM0.NZ
10005718:  b9a0 5488 0144     ST.32 *(ADDR1)(A4+#0x14),M0
                           || SSUBF M2,M1,M0
1000571e:  39a1 0184          ST.32 *(ADDR1)(A4+#0x18),M1
10005722:  be8a 39a2 0023 01c4
                              CMPF  TDM1,M.GT,M1,M3
                           || ST.32 *(ADDR1)(A4+#0x1c),M2
1000572a:  bbc0 7c51 0004     MV.S16   D0,#0x4
                           || ONE   D1
10005730:  3c71 0020          SELECT   TDM1,D0,D1,D0
10005734:  ba70 fd60 fd60 fd60 7d60 0204
                              ST.W8 *(ADDR1)(A4+#0x20),D0
                           || PAD
                           || PAD
                           || PAD
                           || PAD

10005740 <$LBB0_2>:
10005740:  b75e b505 5402 bff8 bff0
```

图 3-5. 普通实现

在 SVGEN 的优化实现中 (如图 3-6 所示) ，利用了 C29 的 QUADF 指令 (通过内嵌函数 __builtin_c29_quadf32 实现) 。该指令将二维空间分成 16 个段。通过 switch() 语句将 16 段空间映射到 6 段空间。C 语言代码如图左侧所示，编译器生成的汇编语言如图右侧所示。生成的汇编代码为直线代码，不含分支，并实现了并行化 (每个周期并行执行四条指令) 。

无论输入如何，在 C29 上，优化实现均需要 24 个周期，而普通实现需要 26-43 个周期，具体取决于输入。在 C28 上，普通实现需要 70-100 个周期。在 Cortex-M7 上，普通实现需要 58-73 个周期，具体取决于输入。



```
__attribute__((section("kernel_opt")))
void svgen(SVGEN_t * v)
{
    float Ubeta = v->Ubeta;
    float Ualpha = v->Ualpha;
    float ScaledUbeta = Ubeta * 0.5f;
    float ScaledUalpha = Ualpha * 0.866f;
    v->tmp1 = Ubeta;
    v->tmp2 = ScaledUbeta + ScaledUalpha;
    v->tmp3 = v->tmp2 - v->tmp1;

    unsigned int index;
    (void)__builtin_c29_quadf32(&index, &ScaledUbeta, &ScaledUalpha);

    float Ta, Tb, Tc;
    int VecSector;
    switch(index) {
      default:
        __builtin_unreachable();
        return;

      case 0:
      case 1:
      case 8:
      case 9:
        Ta = v->tmp2;
        Tb = v->tmp1 - v->tmp3;
        Tc = -v->tmp2;
        VecSector = (index < 8 ? 1 : 4);
        break;

      case 2:
      case 3:
      case 4:
      case 5:
      case 10:
      case 11:
      case 12:
      case 13:
        Ta = v->tmp3 + v->tmp2;
        Tb = v->tmp1;
        Tc = -v->tmp1;
        VecSector = (index < 8 ? 2 : 5);
        break;

      case 6:
      case 7:
      case 14:
      case 15:
        Ta = v->tmp3;
        Tb = -v->tmp3;
        Tc = -(v->tmp1 + v->tmp2);
        VecSector = (index < 8 ? 3 : 6);
```

```
Disassembly of section .kernel_opt:

10005800 <svgen>:
10005800:  9403 b5c0 bea1 4050 b22d 3f5d 0044 3f00
                              MV M3,#0x3f5db22d
                           || LD.32 M0,*(ADDR1)(A4+#0x4)
                           || MV.F16   M1,#5.0E-1
                           || LD.32 M2,*(ADDR2)A4
10005810:  dd01 5c93          SMPYF M4,M0,M1
                           || SMPYF M2,M2,M3
10005814:  bf80 4c54 0c82     QUADF TDM,M3,M4,M2
                           || SADDF M1,M2,M4
1000581a:  bcb6 bbc2 3bc4 0000 0004 0003
                              MV D0,TDM
                           || MV.S16   D2,#0x4
                           || MV.S16   D4,#0x3
10005826:  b9a0 a606 bf60 d488 7c51 0144 0008 00c0
                              ST.32 *(ADDR1)(A4+#0x14),M0
                           || CMP.U16  TDM0,D.LO,D0,#0x8
                           || NEGF  M6,M0
                           || SSUBF M2,M1,M0
                           || ONE   D1
10005836:  b9a1 bcca bf49 3c70 0184 0020 1420 0422
                              ST.32 *(ADDR1)(A4+#0x18),M1
                           || LSL   D0,D1,D0
                           || ADDNEGF  M5,M1,M0
                           || SELECT   TDM0,D1,D1,D2
10005846:  bf60 bbe0 bbc2 6c30 00e1 0303 0006
                              NEGF  M7,M1
                           || AND.U16  D0,#0x303
                           || MV.S16   D2,#0x6
                           || MV D3,D0
10005854:  bbe3 a40b bc70 d4c2 4d11 3c3c 0000 0882
                              AND.U16  D3,#0x3c3c
                           || CMP.S16  TDM1,D.NEQ,D0,#0x0
                           || SELECT   TDM0,D2,D4,D2
                           || SSUBF M3,M0,M2
                           || SADDF M4,M2,M1
10005864:  bf60 bbc0 a473 3f41 0102 0005 0000 0422
                              NEGF  M8,M2
                           || MV.S16   D0,#0x5
                           || CMP.S16  TDM2,D.NEQ,D3,#0x0
                           || SELECT   TDM1,M1,M1,M2
10005874:  b9a2 bbc3 bf41 3f42 01c4 0002 14e5 0881
                              ST.32 *(ADDR1)(A4+#0x1c),M2
                           || MV.S16   D3,#0x2
                           || SELECT   TDM1,M5,M7,M5
                           || SELECT   TDM2,M2,M4,M1
10005884:  bf41 bf42 b9a2 7a09 0c68 04c5 0004
                              SELECT   TDM1,M3,M3,M8
                           || SELECT   TDM2,M1,M6,M5
                           || ST.32 *(ADDR1)(A4+#0x8),M2
                           || RETD
10005892:  bc70 bc71 bf42 39a1 0060 0422 0003 0104
```

图 3-6. 在 C29 上的 SVGEN 优化实现

TI 提供涵盖实时控制和 DSP 的库。在某些情况下，库的优化实现可提升性能，相较于普通实现效果更佳。

3.2.1.4 软件流水线

软件流水线通过利用 C29 CPU 的 VLIW 架构，使环路的多个迭代并行执行。在图 3-7 中，软件流水线在 CFFT 中得到了演示。汇编代码是手写的，充分利用了完整的 128 位指令数据包，每个环路周期内并行执行 8 条指令。

```
sw_pipe_kernel_f3 :
ST.64    *(A6 + A0<<#3), XM08
||SSUBF    M7,  M5, M7
||SADDF    M6,  M5, M7
||LD.64    XM12, *(A4++)
||SSUBF    M15,  M8, M9
||SMPYF    M30,  M20, M24
||SMPYF    M31,  M27, M25


ST.64    *(A6++), XM28
||SSUBF    M14,  M12, M14
||SADDF    M12,  M12, M14
||SADDF    M22,  M22, M23
||SMPYF    M25,  M20, M25
||SMPYF    M24,  M27, M24
||LD.64    XM0, *(A5++A1)
||LD.64   XM2, *(A7++)

ST.64    *(A6 + A0<<#3), XM6
||SSUBF    M15,  M13, M15
||SADDF    M13,  M13, M15
||LD.64    XM20, *(A4++)
||SSUBF    M23,  M16, M17
||SMPYF    M6,  M2, M0
||SMPYF    M7,  M3, M1


ST.64    *(A6++), XM4
||SSUBF    M22,  M20, M22
||SADDF    M20,  M20, M22
||SADDF    M10,  M10, M11
||SMPYF    M1,  M2, M1
||SMPYF    M0,  M3, M0
||LD.64    XM8, *(A5++A1)
||LD.64   XM10, *(A7++)


SDECBD A14, #1, @sw_pipe_kernel_f3
|| ST.64    *(A6 + A0<<#3), XM14
||SSUBF    M23,  M21, M23
||SADDF    M21,  M21, M23
||LD.64    XM28, *(A4++)
||SSUBF    M31,  M24, M25
||SMPYF    M14,  M10, M8
||SMPYF    M15,  M11, M9
```

图 3-7. CFFT 中的软件流水线 - 手写代码

当使用 -O3 优化时，C29 编译器会为 FIR 生成软件流水线代码，如图 3-8 所示。软件流水线加快了环路执行的速度。

```
20107b60 <$L080_8>:
20107b60:  ddbe cc20 c097 c0e8 f64b 79ff
                                SMPYF   M6,M7,M6
                              || SADDF   M0,M4,M0
                              || LD.32   M4,*(ADDR2)(A7++)
                              || LD.32   M7,*(ADDR2)(A4+A0<< 2)
                              || DEC.CIRC    A0,A11
                              || SOECBD  A14,#0x4,$
20107b6c:  bf70 cc69 c0b7 c108 764b 24e4
                                MPYF    M9,M7,M4
                              || SADDF   M1,M5,M1
                              || LD.32   M5,*(ADDR2)(A7++)
                              || LD.32   M8,*(ADDR2)(A4+A0<< 2)
                              || DEC.CIRC    A0,A11
20107b78:  bf70 ccf3 c157 c168 764b 1105
                                MPYF    M4,M8,M5
                              || SADDF   M3,M6,M3
                              || LD.32   M10,*(ADDR2)(A7++)
                              || LD.32   M11,*(ADDR2)(A4+A0<< 2)
                              || DEC.CIRC    A0,A11
20107b84:  bf70 bf48 c0d7 c0e8 764b 156a 0922
                                MPYF    M5,M11,M10
                              || ADDF    M2,M9,M2
                              || LD.32   M6,*(ADDR2)(A7++)
                              || LD.32   M7,*(ADDR2)(A4+A0<< 2)
                              || DEC.CIRC    A0,A11
```

图 3-8. FIR 中的软件流水线 - 编译器自动生成

编译器在 -O3 优化设置下生成软件流水线环路，可提高带有环路的代码的性能。

3.2.2 客户控制和数学运算基准测试

图 3-9 展示了在客户（表示为 A 到 E）提供的某些测试基准中，C29 CPU 与 C28 CPU 在特定任务中的性能对比*（以周期数计）*。这些基准测试涵盖实际客户案例，包括从数学运算到电机控制及插值的多种功能场景。C_Motor 是一项双电机控制基准测试，模拟同时运行两个电机实例的情况。此基准测试中使用了 C29 并行架构，性能*（以周期数计）*比 C28 高 5 倍以上。

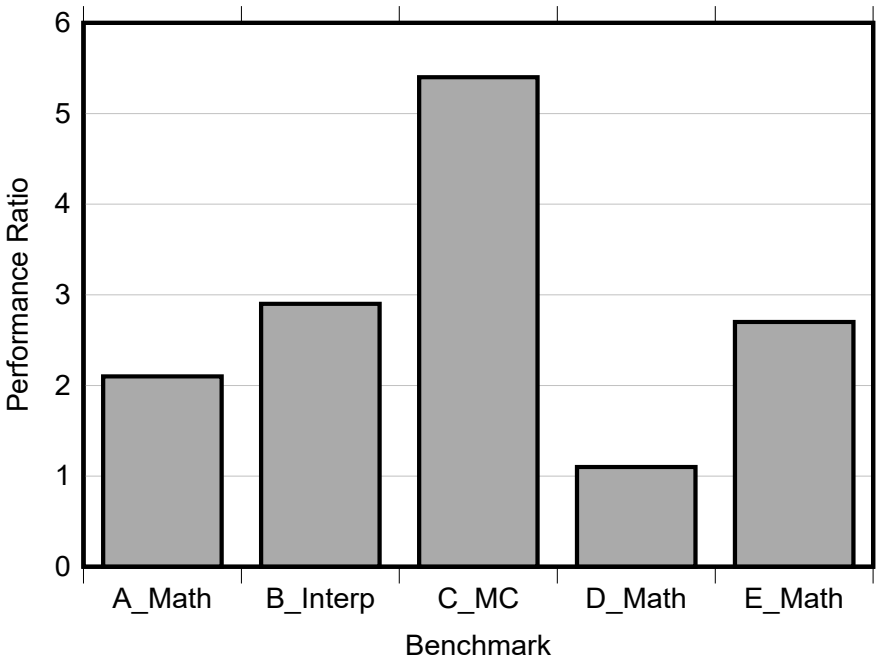


图 3-9. 客户控制和数学运算基准测试

然而，D_Math 测试未能充分发挥 C29 并行架构的优势，因为其代码中大量使用了易变变量。在使用易变变量时，编译器每次需要变量时都会强制从存储器加载或存储到存储器中，而不能将变量保存在寄存器中以减少存储器的访问频率。因此，在实际代码开发中，应谨慎使用易变变量。

3.3 通用处理 (GPP) 性能

除了实时控制功能外，C29 CPU 还具有出色的 GPP 性能。图 3-10 展示了 C29 与 C28 CPU 在客户提供的两项基准测试（分别记为 F 和 G）中的性能对比：F 和 G 基准代表包含 GPP 代码的实际客户基准。F_GPP 基准包含超过 100 个 if () 语句；G_GPP 基准包含超过 30 个 if () 语句。这两个基准测试还包含逻辑、逐位和算术运算。图 3-11 展示了在这些相同基准测试中 C29 与 Cortex-M7 的性能对比，图 3-12 展示了 C29 与某专有 CPU A 的性能对比。在 GPP 代码上，C29 的性能 *（以周期数计）*比 C28 高出近 3 倍、比 Cortex-M7 高出近 50% *（以周期数计）*，比某专有 CPU A 高出近 2 倍（*以周期数计*）。

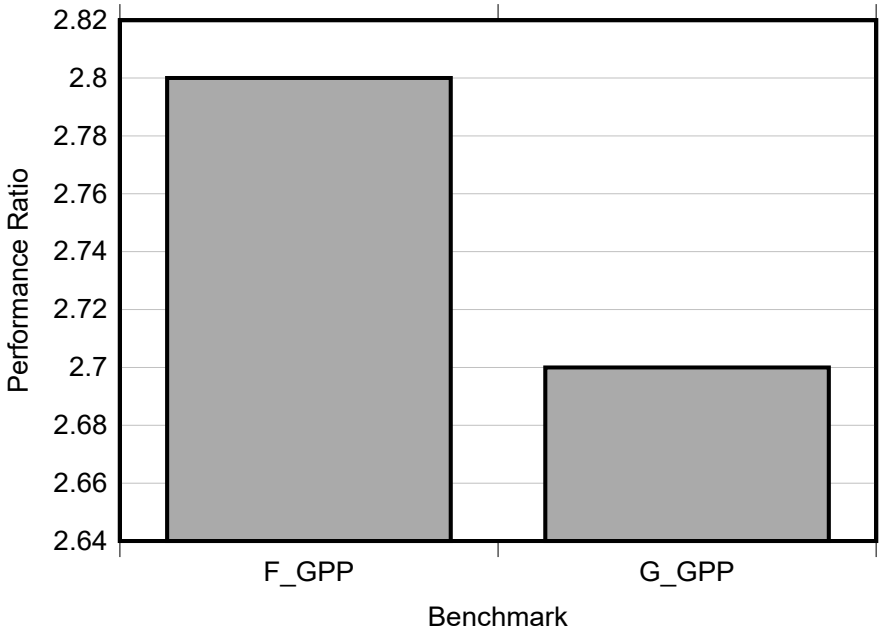


图 3-10. C29 与 C28 的 GPP 性能对比

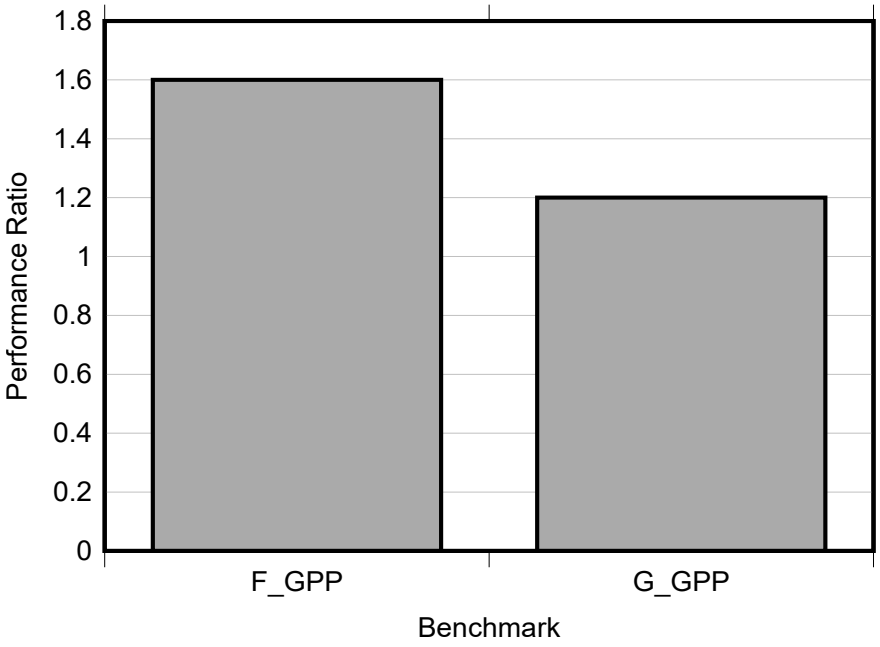


图 3-11. C29 与 M7 的 GPP 性能对比

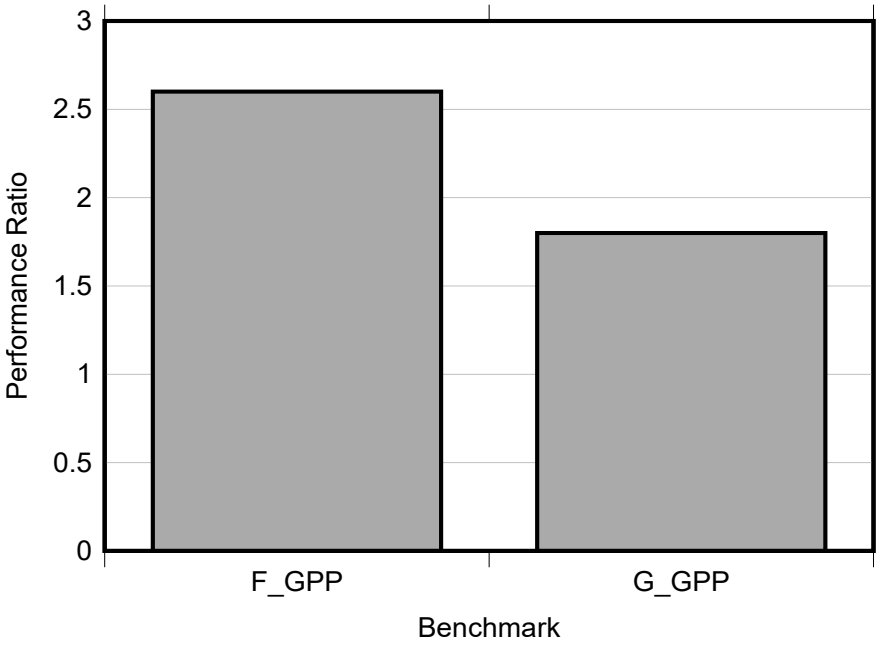


图 3-12. C29 与某专有 CPU A 的性能对比

3.3.1 影响结果的示例和因素

本节提供对于架构和编译器的见解和分析，以解释上面所示的结果。多通用功能单元：

- C29 CPU 内置多个通用功能单元，提升了通用性能。
- 延迟分支指令：实现了几乎无分支延迟的效果，这将在不连续性管理子章节中详细说明。
- 条件执行指令：用于简短的分支处理，例如饱和和死区示例中展示的应用。
- 特殊分支指令：允许 C29 编译器将多个分支目标压缩为一条指令，例如 switch 示例子章节中展示的应用。

3.3.1.1 不连续性管理

传统上，分支 (Branch)、调用 (Call) 和返回 (Return) 操作由于指令流水线的存在会导致开销。在流水线的 Decode-2 阶段，CPU 解析并确定需要执行分支、调用或返回操作。此时，流水线已经填充了后续指令，这些指令需要被清除，然后从不连续目标位置重新抓取指令。清除指令会产生开销。

C29 CPU 具有 9 级流水线，不连续性决策同样发生在 Decode-2 (D2) 阶段。因此，在流水线中，不连续性指令后续的三条指令（Fetch-1、Fetch-2 和 Decode-1 阶段）已经进入执行阶段。除了传统的分支、调用和返回指令外，C29 ISA 还支持*延迟*分支、调用和返回指令（在指令后添加后缀 D，如 CALLD、RETD）。使用延迟指令时，无论不连续性是否发生（如条件分支），其后的三条指令始终会被执行。这三条指令称为*延迟时隙*。C29 编译器在使用延迟指令时，会将适当的指令插入延迟时隙中，从而将原本的三周期不连续开销降低为零周期。

以下两个示例说明了编译器这种结构的使用原理。

- 函数调用：在三个延迟时隙中传递 6 个函数参数。

```
@CALLD  funcA          ; Call funcA
||LD.32 A4,@pointer1   ; Load A4 with pointer1 value from memory
LD.32   A5,@pointer2   ; Load A5 with pointer2 value from memory
||SUB.U16 A6,SP,#34    ; A6 points to value on stack offset -34
MV      A7,#ArrayB     ; Load A7 with address of ArrayB
||LD.32 D0,@variable1 ; Load D0 with Variable1 from memory
LD.32   D1,@variable2 ; Load D1 with Variable2 from memory
; Total Cycles = 4
```

- 函数返回：在延迟时隙中恢复保存的寄存器并释放堆栈空间。

```
funcA: ADD.U16 SP,SP,#24       ; Allocate local stack space
       ST.64   *(SP-#24),XM2   ; Save XM2, XM4, XM6 registers on stack
       ST.64   *(SP-#16),XM4
       ST.64   *(SP-#8),XM6
       ... user code...
       RETD    *(SP-#32)      ; packet 1:Return and restore RPC from stack
       ||MV    M0,M3          ; Place return value in register M0
       LD.64   XM6,*(SP-#8)   ; packet 2:Restore XM6 from stack
       LD.64   XM4,*(SP-#16)  ; packet 3:Restore XM4 from stack
       LD.64   XM2,*(SP-#24)  ; packet 4:Restore XM2 from stack
       ||SUB.U16 SP,SP,#32    ; Deallocate local + return stack space
; Total Cycles = 4
```

以上示例是 C29 编译器如何使用延迟时隙的模型。实际上，延迟时隙不仅仅用于函数实参传递、寄存器恢复和堆栈释放，通常还包含用于实现用户代码的实际功能的指令。

3.3.1.2 Switch() 示例

C29 CPU 特殊的分支指令使编译器能够将多个分支目标折叠为一个指令。*switch* 是通用代码中常见的结构，通常用于处理辅助控制任务。C29 ISA 提供多路分支指令 QDECB 和 DDECB，用于高效实现此语句。四路递减分支 (QDECB) 允许最多四个分支目标，或者选择继续线性执行。四路递减分支 (DDECB) 允许最多两个分支目标，或者选择继续线性执行。

下面的代码块中显示了一个包含 16 个分支的 switch 语句。在 C29 CPU 上，switch 使用一条分支指令 (BCMP) 和四条 QDECB 指令实现，需要 10 到 17 个周期，具体取决于输入。在 Cortex-M7 上，switch 针对每种情况使用比较指令和分支指令实现，需要 6 到 51 个周期，具体取决于输入。

```
switch(state) { case 15: .... break; case 14: .... break; case 13: .... break; ... ... case 0: ....
break; default: .... break; }

C29 Implementation
LD.32 A14,@State
BCMP @default,A.GT,A14,#15    QDECBA14,#0x4,@case15,@Case14,@Case13,@Case12,@
QDECBA14,#0x4,@case11,@Case10,@Case9,@Case8,@    QDECBA14,#0x2,@case7,@case6,@case5,@case4,@
QDECBA14,#0x2,@case3,@case2,@case1,@case0,@
default:
```

```
....
....
LB @State_end
case15:
....
....
LB @State_end
case14:
....
....
LB @State_end
case13:
....
....
LB @State_end
....
....
....
case2:
....
....
LB @State_end
case1:
....
....
LB @State_end
case0:
....
....
State_end:

M7 Implementation
LDRSB R6,[State]
CMP   R6,#15
BGT.N default
BEQ.N case15
CMP   R6,#14
BEQ.N case14
....
CMP   R6,#0
BEQ.N case0
default:
....
....
B  State_end
case15:
....
....
B  State_end
case14:
....
....
B  State_end
case13:
....
....
B  State_end
....
....
....
case2:
....
....
B  State_end
case1:
....
....
B  State_end
case0:
....
....
State_end:
```

3.4 基于模型的设计基准测试

客户越来越倾向于选择基于模型的设计和自动代码生成，因此，了解自动代码生成工具（如 MathWorks 的 Embedded Coder）的性能非常重要。在本文发布时，Embedded Coder 的已发布版本尚不支持 C29，因此基准

测试使用的是为 C28 CPU 生成的 C 代码。基于无传感器磁场定向控制的电机控制模型包含闭环控制和滑模观测器 (SMO)。生成的代码包含实时控制元件以及 GPP 元件。基于模型的设计基准测试 展示了基准测试结果。结果表明 C29 的性能 *（以周期数计）* 比基于 Cortex-M4 的竞品 MCU 的性能要高出 2 倍以上。

表 3-2. 基于模型的设计基准测试

MCU	周期	性能比率
#6 (Cortex-M4)	877	1
F29H85x (C29)	393	2.23
F29H85x (C29)	312（对生成的代码进行了一些手动优化）	2.81

3.5 应用基准测试

到目前为止，我们展示了以下基准测试的结果：实时信号链、客户提供的基准测试、特定控制和 DSP 模块，以及通用基准测试。本节对比了使用 C29 和 C28 的性能结果，这些基准测试基于 C2000 参考设计，主要关注于实时应用场景，如数字电源和电机控制。

目前，基于 C29 的参考设计尚未发布，仍处于开发阶段，因此本文仅展示早期基准测试结果。

3.5.1 单相 7kW OBC 说明

TIDM-2013 是使用 F28x 系列器件构建的参考设计，用于实现单相 OBC 设计。此设计由交错式连续导通模式 (CCM) 图腾柱 (TTPL) 无桥功率因数校正 (PFC) 功率级和 CLLLC 直流/直流功率级组成。此设计运行以下 ISR 的对应频率：

- ISR1（PWM 更新）：120KHz
- ISR2（PFC 电流、CLLC 电压环路）：120KHz
- ISR3（PFC 电压环路，仪表采集）：10KHz

图 3-13 展示了 F29x 与 F28x 在 200MHz 相同 CPU 时钟频率执行上述 ISR 的基准测试。ISR1：F29x 与 F28x 性能无变化，因为此 ISR 的主要操作是 PWM 外设寄存器写入，两者实现相同。ISR2：在 F29x 上的执行速度比 F28x 快 1.7 倍。

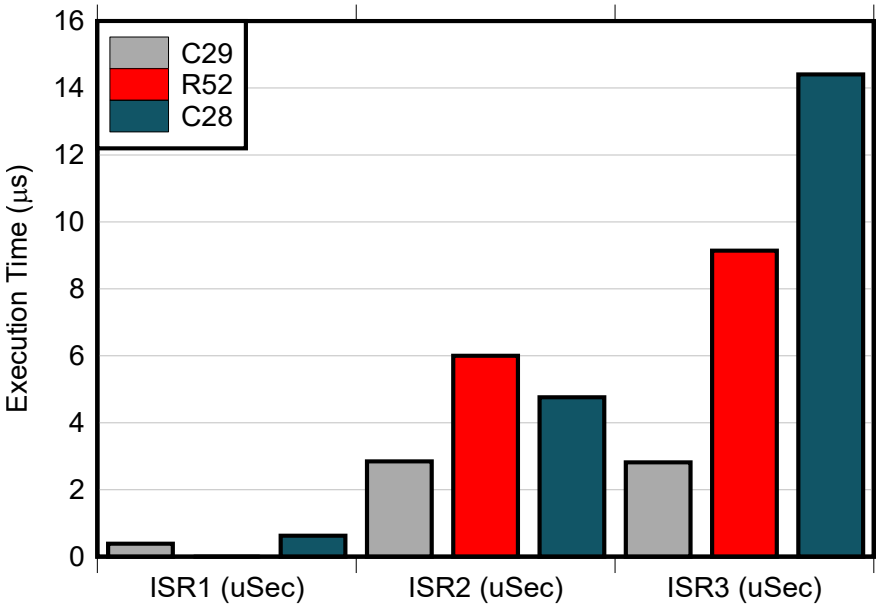


图 3-13. OBC 基准

3.5.2 基于 Vienna 整流器的三相功率因数校正

高功率三相功率因数 (AC-DC) 应用中（例如非板载 EV 充电器和通信电源整流器）使用了 Vienna 整流器电源拓扑。TIDM-1000 说明了使用 C2000™ 微控制器 (MCU) 控制功率级的方法。TIDM-1000 基准测试 展示了实验 4（闭环电压控制，带内环电流控制和中点电压平衡）中的早期基准测试结果。在 ISR 中测量从开始到结束的周期。测试结果表明，F29x 的性能 *（以周期数计）* 是 C28 的 2 倍。

表 3-3. TIDM-1000 基准测试

TIDM-1000	周期	性能比率
F2837x (C28)	308	1
F29H85x (C29)	153	2.01

3.5.3 单相位逆变器

TIDM-HV-1PH-DCAC 使用 C2000™ F2837xD 和 F28004x 微控制器来实现单相逆变器（直流/交流）控制。TIDM-HV-1PH-DCAC 基准测试 展示了实验 3（闭环电压控制，带内环电流控制）中的早期基准测试结果。在 ISR 中测量从开始到结束的周期。测试结果表明，C29 的性能 *（以周期数计）* 比 C28 提升了 80%。

表 3-4. TIDM-HV-1PH-DCAC 基准测试

TIDM-HV-1PH-DCAC	周期	性能比率
F2837x (C28)	609	1
F29H85x (C29)	332	1.83

3.5.4 机器学习

实时控制中的机器学习 (ML) 技术不断涌现，应用领域包括电弧故障检测和电机故障检测等。尽管在运行嵌入式 AI 模型方面，片上人工智能 (AI) 加速器逐渐普及，但实时控制 CPU 的 ML 性能也非常重要。机器学习基准测试 展示了在 Cortex-M7 MCU 和基于 C29 的 F29H85x MCU 上进行 3 层、4 层和 5 层计算神经网络 (CNN) 的基准测试。即使在 Cortex-M7 的工作频率是 C29 的 2 倍的情况下，C29 仍然比 Cortex-M7 快将近 5 倍。

表 3-5. 机器学习基准测试

型号	Cortex-M7 400MHz、浮点模型（毫秒）	F29H85x (C29) 200MHz、浮点模型（毫秒）
3 层 CNN	11.54	2.33
4 层 CNN	11.82	2.35
5 层 CNN	12.02	2.30

3.6 闪存存储器效率

闪存执行效率至关重要，因为并非所有代码都可以从零等待状态存储器运行。在 F29H85x 上，以 200MHz 的工作频率运行时，访问闪存需要历经 3 个等待状态。为了减轻等待状态的影响，F29H85x 提供了预取机制和块缓存机制，并默认启用。闪存效率基准测试 展示了多个基准测试中 F29H85x 和 F2837x（同样需要历经 3 个等待状态，运行在 200MHz）的闪存效率对比结果（单位为 %）。在很多基准测试中，从闪存运行与从零等待状态存储器运行类似。

表 3-6. 闪存效率基准测试

基准	F2837x (%)	F29H85x (%)
CFFT	81	93
FIR	91	86
IIR（环路）	82	99
信号链 (ACI)	95	97
二进制 LUT 搜索	82	97

3.7 代码尺寸效率

除了性能效率外，代码尺寸效率也是一个重要指标，在零等待状态内存资源有限的情况下尤为重要。性能关键的代码通常在零等待状态内存中运行，而非性能关键的代码则在闪存存储器中运行。代码尺寸基准测试 展示了各种基准测试的代码尺寸效率，并将 C29 与 C28 和 ARM (Cortex-M7) 进行了比较。从结果中可以注意到以下几点：

- C29 的代码尺寸大体与 C28 以及 Cortex-M7 相当。某些基准测试中，C29 的代码尺寸较小，而在另一些测试中，则略大。
- 代码尺寸效率与编译器的 -O3 优化设置相对应。用户可以灵活地对部分代码选择性地使用 -Oz 以减小代码尺寸。
- C29 FIR 的代码尺寸较大，原因是使用了软件流水线，此举可以显著提升性能。整体来看，环路代码只占整体代码的一小部分。

表 3-7. 代码尺寸基准测试

DSP、数学和实时基准测试	C28 与 C29 的代码尺寸：(C) <1 时，表示 C28 的代码尺寸更小	Cortex-M7 与 C29 的代码尺寸：(C) <1 时，表示 Cortex-M7 的代码尺寸更小
FIR	0.5	0.7
IIR	0.7	1.4
DCL - 数字控制库	1.5	1.5
FCL - 快速电流环路	1.1	1.1
SPLL - 软件锁相环	1.7	1.2
SVGEN	1.2	1
ACI 信号链	0.9	0.6
客户 DSP、数学和实时基准测试		
B_Interp	1.1	1
C_Motor	1.3	1
D_Math	0.8	0.8
E_Math	1	0.7
GPP 基准		
F_GPP	0.8	0.8
G_GPP	0.7	0.7
参考设计		
Vienna 整流器	0.94	–
单相位逆变器	0.75	–

4 总结

工业和汽车应用对效率和功率密度的要求不断提高，因此对具有更高性能、可扩展的实时 MCU 的需求也十分旺盛。这些 MCU 应能够支持先进的拓扑和集成选项，并提供内置的功能安全和信息安全功能。全新 C29 CPU 拥有卓越的实时性能，专为应对这些挑战而优化。C29 CPU 的并行架构支持在单个内核中实施传统上需要多个 CPU 的功能。本白皮书通过广泛的基准测试验证了 C29 CPU 的强大能力。C29 编译器可直接为 C 代码提供出色的性能支持。与 C29 CPU 紧密耦合的 SSU，使用户无需重新编程即可轻松开发符合 ASIL-D 标准的安全应用。

5 参考资料

1. 德州仪器 (TI)，F29H85x 和 F29P58x 实时微控制器数据表
2. 德州仪器 (TI)，F29H85x 和 F29P58x 实时微控制器技术参考手册
3. 德州仪器 (TI)，将应用软件迁移到 C29 CPU 用户指南
4. 德州仪器 (TI)，使用 C29x SSU 实现运行时功能安全和信息安全保护应用手册
5. 德州仪器 (TI)，TI C29x Clang 编译器工具用户指南
6. 德州仪器 (TI)，展示 C2000 控制 MCU 优化信号链的实时基准测试应用手册
7. 德州仪器 (TI)，从 TMS320F2837x、TMS320F2838x、TMS320F28P65x 迁移到 TMS320F29H85x

重要声明和免责声明

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 产品的应用。严禁对这些资源进行其他复制或展示。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