Technical White Paper

The C29 CPU – Unrivaled Real-Time Performance with Optimized Architecture on C2000™ MCUs

Saya Goud Langadi, Chen-Yu Chang, Sira Rao

ABSTRACT

To meet emerging design trends for higher power density and complex control techniques in real-time applications, engineers need high-performance MCUs with more flash memory, larger computational capabilities, and higher levels of integrated functionality. Requirements are enabled through innovations in CPU architecture, such as the C29 CPU in TI C2000™ MCUs that has 64-bit architecture and advanced cybersecurity components, such as the safety and security unit (SSU). The SSU allows context isolation among threads running within the same CPU, enabling run-time security and freedom from interference (FFI), a feature typically found in microprocessors. The C29 core builds on TI's market leading C28 core, delivering higher performance for general purpose and digital signal processing.

This white paper discusses the C29 core architecture, benefits of the SSU, and describes several performance benchmarks comparing MCUs with C29 cores to MCUs with other CPU core architectures. The paper describes the benefits of the parallel C29 architecture, and the performance entitlement achieved with the C29 compiler.

Table of Contents

List of Figures

List of Tables

Trademarks

1 Introduction to Real-Time Control

In real-time control, a closed-loop system gathers data, processes it in a control loop, and makes updates within a defined time window. Signal chain performance quantifies real-time control performance, where higher performance enables faster closed-loop systems. A real-time control system is typically composed of three main elements:

- **Sensing or feedback acquisition**: The application can require the accurate measuring of key parameters (such as voltage, current, motor speed, motor position, and temperature) at precise moments in time.
- **Processing and Control**: Use sensor data to apply control algorithms on incoming data to compute the next output command.
- **Actuation**: The application of the calculated output command to the system to control the output. Changing the duty cycle of a pulse width modulator (PWM) unit driving a power electronics system is an example of actuation.

In real-time control, performance of the system is determined not just by the processing power of the CPU, but also how fast peripherals are accessed the speed of the interrupt response. These factors lead to the notion of a real-time signal chain.

Figure 1-1 illustrates various components involved in a typical real-time signal chain of motor control and digital power systems. Better signal chain performance enables higher DC bus use and the operating speed range of a motor in motor control applications. In digital power applications, better signal chain performance enables higher control loop frequencies leading to smaller components and lower cost.

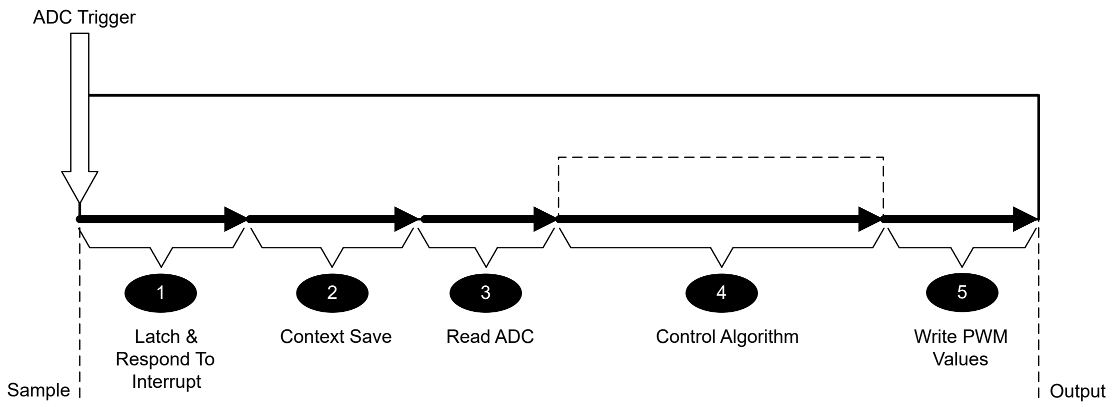


Figure 1-1. Real-Time Signal Chain Components

Components 1, 2, and 4 are dominated by the CPU architecture while components 3 and 5 are dependent on CPU and device architecture. This paper primarily highlights improvements in components 1, 2 and 4. Additionally, C2000 MCUs offer low latency interconnect, enabling single-cycle ADC reads and single-cycle PWM updates.

2 C29 CPU and Key Features

Figure 2-1 shows the block diagram of the C29 CPU, and its key features and benefits are highlighted below.

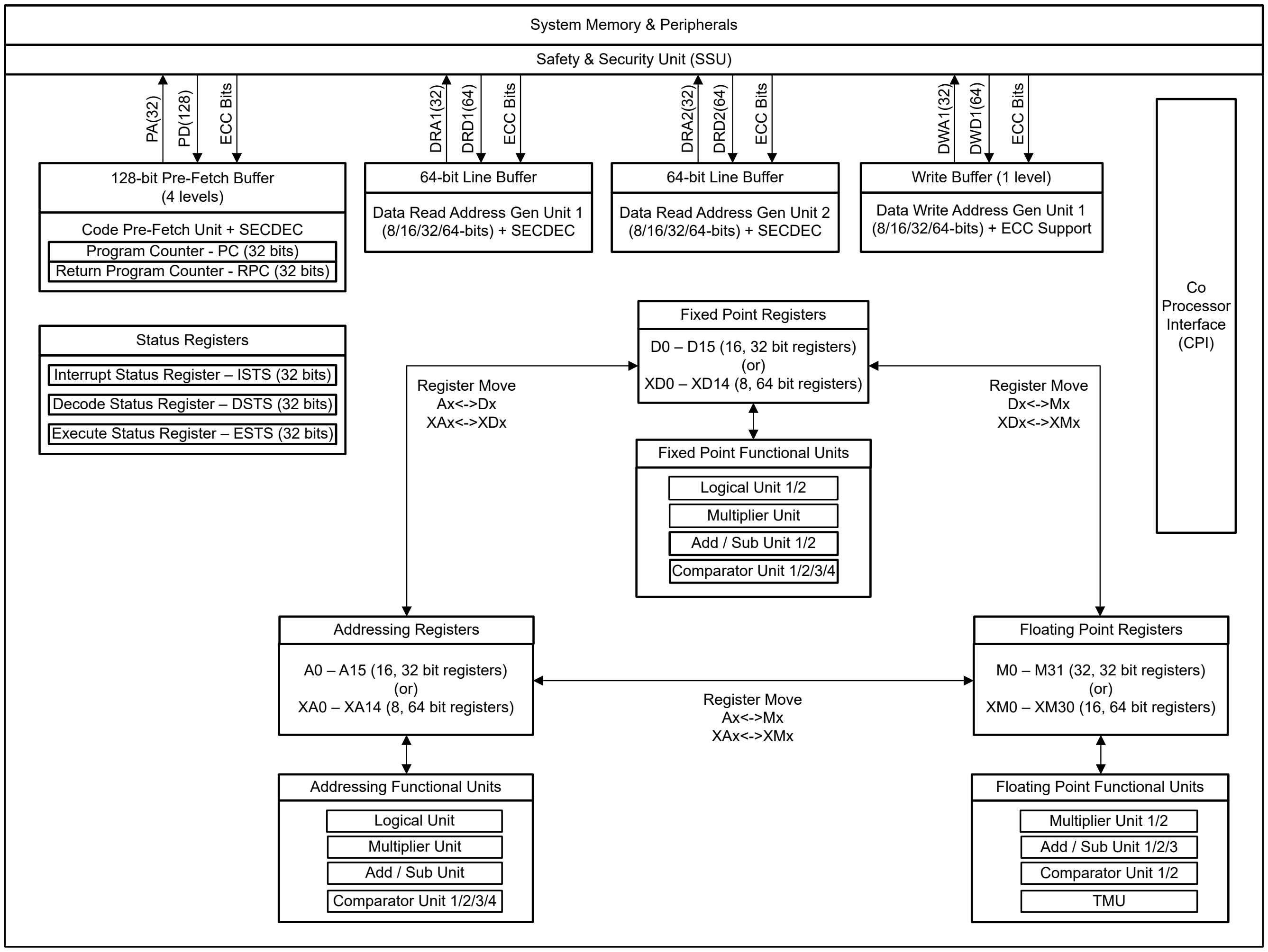

System Memory & Peripherals
Safety & Security Unit (SSU)
PA(32)
PD(128)
ECC Bits
DRA1(32)
DRD1(64)
ECC Bits
DRA2(32)
DRD2(64)
ECC Bits
DWA1(32)
DWD1(64)
ECC Bits
128-bit Pre-Fetch Buffer (4 levels)
Code Pre-Fetch Unit + SECDEC
Program Counter - PC (32 bits)
Return Program Counter - RPC (32 bits)
64-bit Line Buffer
Data Read Address Gen Unit 1 (8/16/32/64-bits) + SECDEC
64-bit Line Buffer
Data Read Address Gen Unit 2 (8/16/32/64-bits) + SECDEC
Write Buffer (1 level)
Data Write Address Gen Unit 1 (8/16/32/64-bits) + ECC Support
Co Processor Interface (CPI)
Status Registers
Interrupt Status Register – ISTS (32 bits)
Decode Status Register – DSTS (32 bits)
Execute Status Register – ESTS (32 bits)
Fixed Point Registers
D0 – D15 (16, 32 bit registers) (or) XD0 – XD14 (8, 64 bit registers)
Register Move Ax<->Dx XAx<->XDx
Register Move Dx<->Mx XDx<->XMx
Fixed Point Functional Units
Logical Unit 1/2
Multiplier Unit
Add / Sub Unit 1/2
Comparator Unit 1/2/3/4
Addressing Registers
A0 – A15 (16, 32 bit registers) (or) XA0 – XA14 (8, 64 bit registers)
Register Move Ax<->Mx XAx<->XMx
Floating Point Registers
M0 – M31 (32, 32 bit registers) (or) XM0 – XM30 (16, 64 bit registers)
Addressing Functional Units
Logical Unit
Multiplier Unit
Add / Sub Unit
Comparator Unit 1/2/3/4
Floating Point Functional Units
Multiplier Unit 1/2
Add / Sub Unit 1/2/3
Comparator Unit 1/2
TMU

Figure 2-1. C29 Architecture Block Diagram

VLIW CPU: The C29 is based on a Very Long Instruction Word (VLIW) architecture. Variable size instructions (16-bit, 32-bit, and 48-bit) are supported. The size of the instruction packet can be 16-bit to 128-bits, thus enabling better code density, as well as up to eight 16-bit instructions that are executed in a single CPU cycle.

CPU Memory Bus: A 128-bit wide program bus can fetch a 128-bit wide instruction packet for the CPU to execute. Two 64-bit read buses enable parallel reads of 64-bits, and a 64-bit write bus enables writing 64-bit data to memory, all in a single cycle.

Byte addressability and Data Types: The C29 supports byte addressing, with data types fully compatible with other popular CPU architectures such as ARM.

CPU Registers: There are three sets of registers; Ax, Dx, and Mx. Ax registers (16 32-bit registers A0-A15 or eight 64-bit registers XA0-XA14) are primarily meant for address generation. Additionally, certain integer operations are executed in the early phase of pipeline for improved performance. Dx registers (16 32-bit registers D0-D15 or eight 64-bit registers XD0-XD14) are meant for integer fixed-point operations and Mx registers (32 32-bit registers M0-M31 or 16 64-bit registers XM0-XM30) are for floating-point operations.

Functional Units: There are a total of 24 functional units associated with register sets Ax, Dx, and Mx, and special function registers. Each functional unit supports a set of instructions. Certain functional units have multiple instances. As an example, there are four compare units associated with the Ax register file that evaluate two cases of a switch statement every cycle that improves switch statement execution. There are two floating-point multiply and three floating-point add or subtract units associated with the Mx register set that execute one FFT butterfly every two cycles.

Trigonometric Math Unit (TMU): Trigonometric operations are supported and extended for a 64-bit dual precision floating-point, in addition to a 32-bit single precision floating-point.

Interrupts: The C29 supports regular interrupts (termed INT) and an optimized interrupt called real-time interrupt (RTINT). RTINT uses a dedicated hardware interrupt stack. When an RTINT occurs, CPU context is saved off automatically to this stack, which is faster than a software based context save mechanism). In addition to being faster, it is also a fixed number of cycles - thus improving determinism - whereas a software based context save mechanism can take a variable number of cycles. Hardware interrupt prioritization is supported to reduce software overhead of prioritization through software.

Safety: Higher ASIL levels requires code isolation among multiple threads running within or across CPUs. The Safety and Security Unit (SSU) enables isolation among these threads. In a simplistic form, SSU allows a user to define multiple associated memory regions (called Access Protected Regions (APR)) that can be tied together (through a concept known as a LINK) to create an isolated thread. A thread consists of code, data, a stack, and peripherals. A specific code LINK can access specific data LINKs through Read, Write, or both Read and Write permissions. The advantage of SSU over a traditional MPU is that permissions are enforced based on code being executed. As a result, there is no need to reprogram the MPU. Each thread has a hardware STACK and STACKs are switched automatically in the CPU to enable full isolation. In an OS context - AUTOSAR for example - this efficient switching allows real-time ISRs to be CAT1 interrupts unaffected by the OS, and be completely isolated from the AUTOSAR application. Thus, a single C29 CPU core can run an OS and control tasks without affecting control performance.

Security: When code execution moves across STACKs, entry and exit points are enforced. Entry and exit points are well-defined points where one thread calls or branches into, or returns from, another thread. Calling, branching to, or returning from any other address creates an exception, therefore avoiding security attacks. The SSU also supports firmware updates and debug through a mechanism called ZONEs, with each ZONE having independent password and debug settings. ZONEs enable secure multiparty development, where each party defines a password to block code visibility and controls code debug by another party.

Table 2-1. C29 Major Feature

Feature	Comment
Ease of Use	• Byte addressable CPU • Linear and unified memory map with 4GB address range • Fully protected pipeline • Deterministic execution without cached memories
Improved Parallelism	• Execute 1 to 8 instructions in parallel • Execute fixed-point, floating point, and addressing operations in parallel • Specialized instructions for decision making code and real-time control (example: if-then-else statements, trigonometric and multiphase vector translation operations)
Improved Bus Throughput	• Capable of fetching up to 128-bit instruction word every cycle • Capable of performing 8, 16, 32, 64-bit dual reads and single writes per cycle • Improved addressing modes which reduce overhead in accessing memory and peripheral resources
Code Efficiency	• Supports variable length instruction set (16-bit, 32-bit, and 48-bit) • Critical operations are encoded as 16-bit and 32-bit opcodes for improving the code density • Rich instruction set optimizes operations in smallest instructions
ASIL-D Safety Capability	• Support for both Lockstep and split lock modes • Integrated ECC logic enables end to end safe interconnect • Separate code threads can be fully isolated including stack using SSU • Zero CPU overhead switching from one thread to other in HW automatically enabled best real-time performance
Multi-zone Security	• Run time content protection and IP protection of code • Individual passwords for each zone to control access
Enhanced Debug and Trace Capabilities	• Specialized data logging and code flow trace instructions • Trace data capable of being logged in on-chip RAM or exported through serial communication peripherals

2.1 Parallel Architecture and Compiler Entitlement

The C29 ISA has specific designs and instructions targeted to improving specific performance characteristics, such as:

Real-Time Control and General Purpose Processing

MINMAXF: The MINMAXF instruction bounds a floating-point value present in an M register to a lower limit and an upper limit, specified in two other M registers.

QUADF: The QUADF instruction sets TDM register (a CPU status register) flags to break up the two dimensional vector system into 16 segments. By using a scaled value of the input co-ordinate values, the TDM flag results identifies the segment in a six-segment space-vector generation method. This approach can be extended to other space-vector variants as well.

Minimize Discontinuities for Decision Making Code

XC: The XC conditional execute instruction checks appropriate status flags A.Z, A.N, A.ZV and A.Z in the DSTS register, based on the selected instruction. Based on the flags value, a set of instruction packets either execute instructions or function as a NOP (no operation).

SELECT: The SELECT instruction uses a test condition to select between one of two source registers (such as A, D, or M registers), whose contents are then copied to a destination register of the same type.

Special Branches

The CPU supports a concepted called Delayed branches, which help achieve zero overhead on discontinuities (this is explained further below).

Branch instructions with conditions based on LUT functions using test flags (TA.MAP, TDM.MAP) are supported.

QDECB: The multiway QDECB conditional branch instruction checks the content of the A14 register. Based on the value of the A14 register, program execution either branches to one of four designated branch destinations or continues with the next instruction packet with a decrement of the A14 register.

DDECB: The multiway DDECB conditional branch instruction checks the content of the A14 register. Based on the value of the A14 register, program execution either branches to one of two designated branch destinations or continues with the next instruction packet with a decrement of the A14 register.

The advanced C29 compiler selects the appropriate instructions above, based on the need. In some cases, such as QUADF, use built-in intrinsics in C code to leverage the corresponding instruction and obtain optimized performance.

The instruction set user guide mentions the built-in intrinsic (if available) corresponding to an instruction.

3 C29 Performance Benchmarks

The C29 CPU is designed to offer at least double the performance of the C28 CPU at the same operating frequency. This section presents performance benchmarking results of the C29 versus C28 and competition CPUs, and in each case presents some analysis and insight into what aspects of the architecture and compiler enable that performance level.

Unless otherwise mentioned, benchmarking occurred with the compiler settings optimized for speed (-O3 for C29 compiler) and run from zero wait-state memory. C29 benchmarking results can update over time, with updates to the C29 compiler. Current results are with the 0.1.0.STS version of the compiler.

Similarly, for competition devices, benchmarking occurred with compiler settings optimized for speed, and run from zero wait-state memory.

3.1 Signal Chain Benchmark with ACI Motor Control

The ACI Motor Control Benchmark simulates the sensorless AC induction (ACI) motor control application. The application performs all the typical operations, including analog-to-digital converter (ADC) reads for sensing phase currents, transforming blocks that operate on the sensed current, and PWM writes to control phase voltages. No special external hardware is needed to provide stimulus as a block of code in the application models the behavior of an induction motor. To simulate closed loop behavior, the expected current from the motor model is fed into the ADC through the DAC modules. A single ADC is configured to sense the phase A and phase B currents sequentially through two channels. Phase C current is derived from phase A and phase B currents and is not sensed. Three PWM writes simulate control of duty cycle of the three phase A, B, and C voltages.

Figure 3-1 represents the execution blocks in the control loop interrupt routine of the benchmark application. The control loop interrupt is triggered at a rate of 2KHz and 1024 iterations of the control loop interrupt routine are executed before the application terminates. The "ACI Model" and "Inverse Clarke and DAC output" blocks represent code blocks that are not part of a real ACI motor control application, but are used in the benchmark for simulating the behavior of the motor.

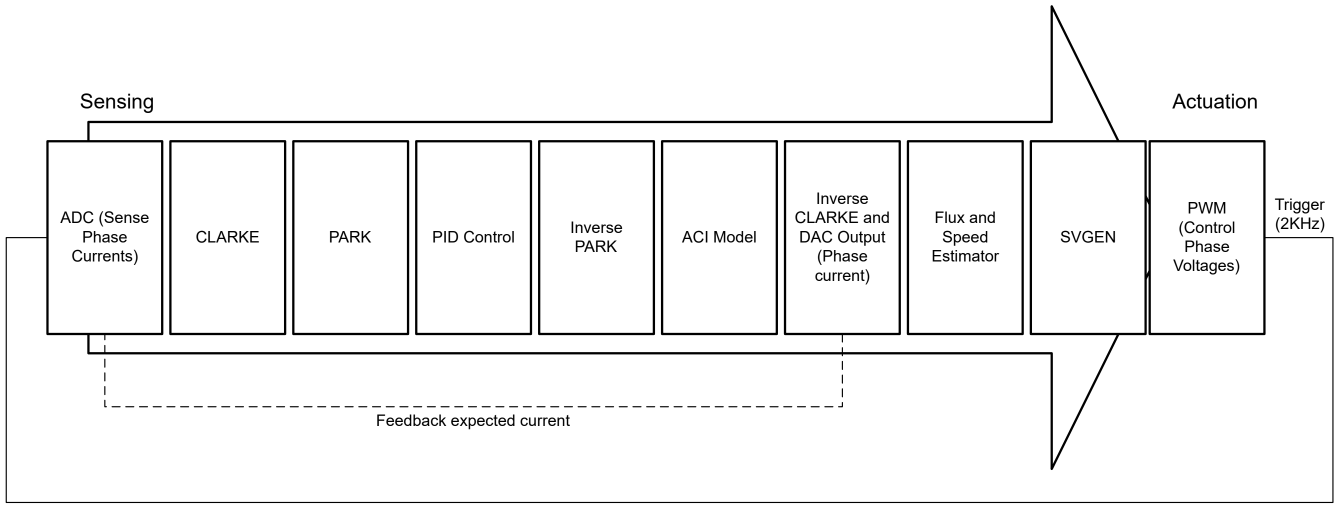

Sensing
Actuation
ADC (Sense Phase Currents)
CLARKE
PARK
PID Control
Inverse PARK
ACI Model
Inverse CLARKE and DAC Output (Phase current)
Flux and Speed Estimator
SVGEN
PWM (Control Phase Voltages)
Trigger (2KHz)
Feedback expected current

Figure 3-1. Real-Time Control Loop

Signal Chain Performance of Real-Time Control MCUs summarizes the real-time signal-chain performance of various competition MCUs targeted for real-time control applications. The results include several notable points:

- The F29H85x with a C29 CPU takes the lowest CPU cycles to run the signal chain benchmark, compared to C28 and competition MCUs.
- The F29H85x with a C29 CPU is 4.31 times faster (cycles) than a competition MCU (*1*) with a Cortex-M7 CPU
- Even though the F29H85x runs at 200MHz, if the competition MCU *1* at 480MHz is considered as the baseline, the effective speed per CPU core (eMHz/Core) for the F29H85x is 862MHz (4.31 x 200). Competition MCU *1* needs to run at 862MHz in order to match the signal-chain performance of the F29H85x running at 200MHz.

Table 3-1. Signal Chain Performance of Real-Time Control MCUs

MCU	CPU	CPU type	CPU frequency	Accelerator	Cycles	Perf. ratio	eMHz/Core
1	Cortex-M7	6-stage superscalar pipeline, branch prediction	480	–	1094	1	480
2	Cortex-M4	3-stage pipeline, branch prediction	170	CORDIC	838	1.30	220
3	Proprietary A	4-stage superscalar pipeline (dual-issue), branch prediction	300	–	857	1.28	384
4	Proprietary B	5-stage pipeline, limited dual-issue	200	TFU	894	1.22	244
5	Proprietary C	5-stage pipeline	240	–	1295	0.84	202
AM263P	Cortex-R5F	8-stage pipeline, limited dual-issue, branch prediction	400	TMU	705	1.55	620
F2837x	C28	8-stage pipeline, limited dual-issue	200	TMU	527	2.08	416
F29H85x	C29	9-stage pipeline VLIW (up to 8 instructions)	200	TMU	254	4.31	862

3.2 Real-time Control and DSP Performance

The C29 CPU is very efficient at performing real-time control and DSP operations. Detailed benchmarking demonstrates this capability on the following benchmarks:

- CFFT - Complex Fast Fourier Transform
- FIR - Finite Impulse Response Filter
- IIR_sample - one input sample of an Infinite Impulse Response filter
- IIR_loop - a block of input samples of an Infinite Impulse Response filter
- DCL - Digital Control Library (comprising PI, PID, etc.)
- FCL - Fast Current Loop
- SPLL - Software Phase Locked Loop
- SVGEN - Space Vector Generation
- FOC - Field Oriented Control for Motor Control (same as ACI signal chain)
- Bin_LUT - Binary LUT search

Figure 3-2 shows C29 versus C28 performance on the above benchmarks. On average, considering the benchmarks shown, the C29 is 3x better *(in cycles)* than the C28 CPU.

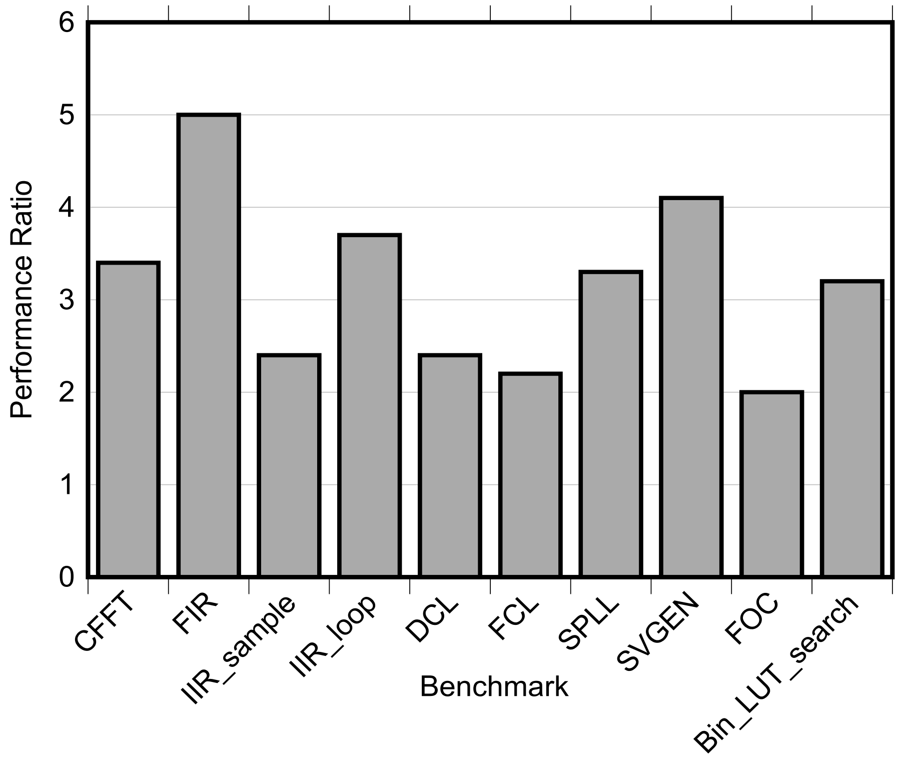


Figure 3-2. C29 versus C28 Real-time Control and DSP Performance

Figure 3-3 shows C29 versus Cortex-M7 performance on the above benchmarks. On average, considering the benchmarks shown, the C29 is almost 4x better **(in cycles)** than the Cortex-M7 CPU.

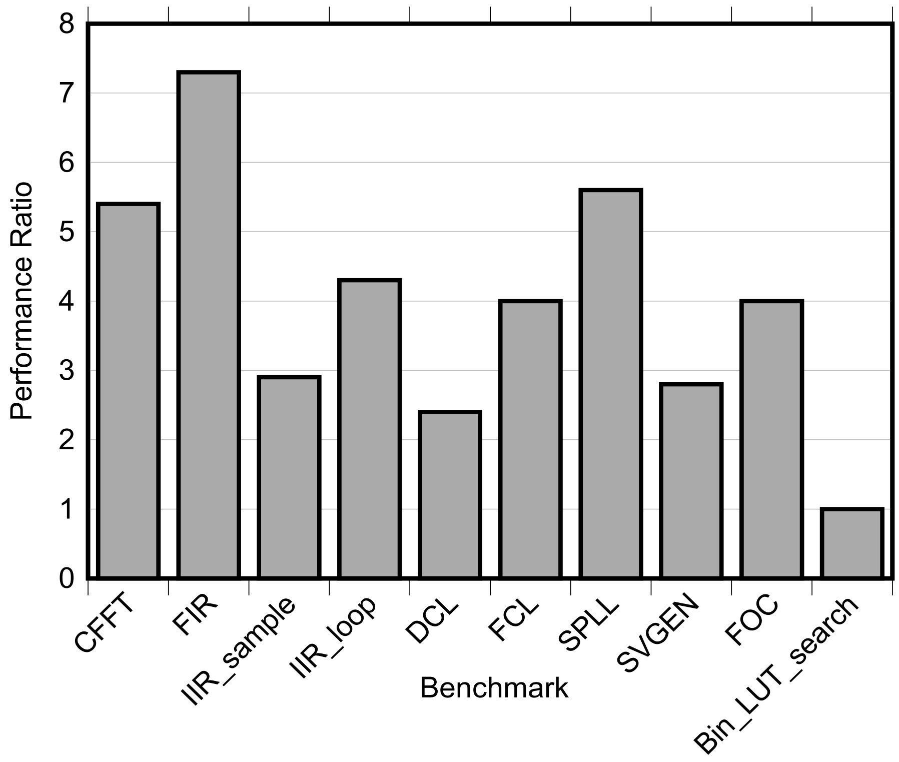


Figure 3-3. C29 versus M7 Real-time Control and DSP Performance

Figure 3-4 shows C29 versus a Proprietary CPU performance on the above benchmarks. On average, considering the benchmarks shown, the C29 is four times better *(in cycles)* than the popular Proprietary CPU.

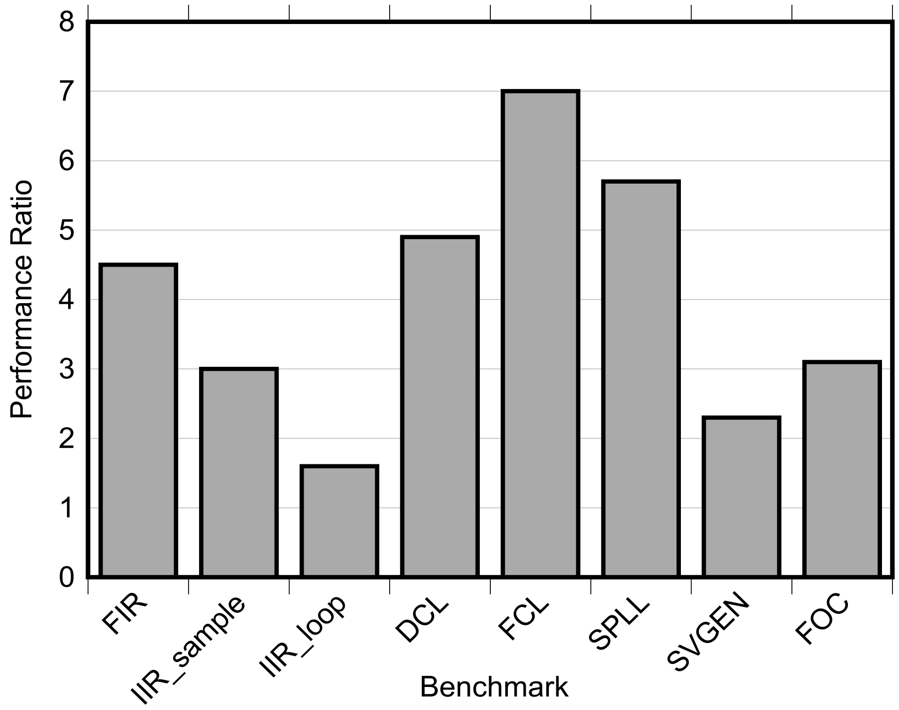


Figure 3-4. C29 versus Proprietary CPU A Real-time control and DSP Performance

3.2.1 Examples and Factors Contributing to Results

This section provides insight and analysis into the architecture and compiler to help understand the results illustrated above.

3.2.1.1 Saturation (or Limiting) Example

Saturation type code commonly occurs in real-time applications. shows a summary of two different ways saturation is implemented in C.

The code block below shows the if..else based approach to implementing saturation. The C29 (11 cycles, independent of input) outperforms the Cortex-M7 (14-27 cycles, dependent on input). On the C29, the if() is implemented through a conditional branch instruction (BC), and for the remaining two paths (the elseif and else), a compare (CMPF) followed by a conditional instruction (XCP) is used, thus avoiding branches.

```
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:
```

The code block below shows the ternary operator '?' based approach to implementing saturation. The C29 (three cycles, independent of input) outperforms the Cortex-M7 (18-22 cycles, dependent on input), through the MINMAXF instruction without any branches.

```
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]
```

The C29 compiler is enhanced to generate equal performance regardless of the if..else or the ternary operator based approaches.

|| denotes instructions occurring in parallel with the above instructions.

3.2.1.2 Dead Zone Example

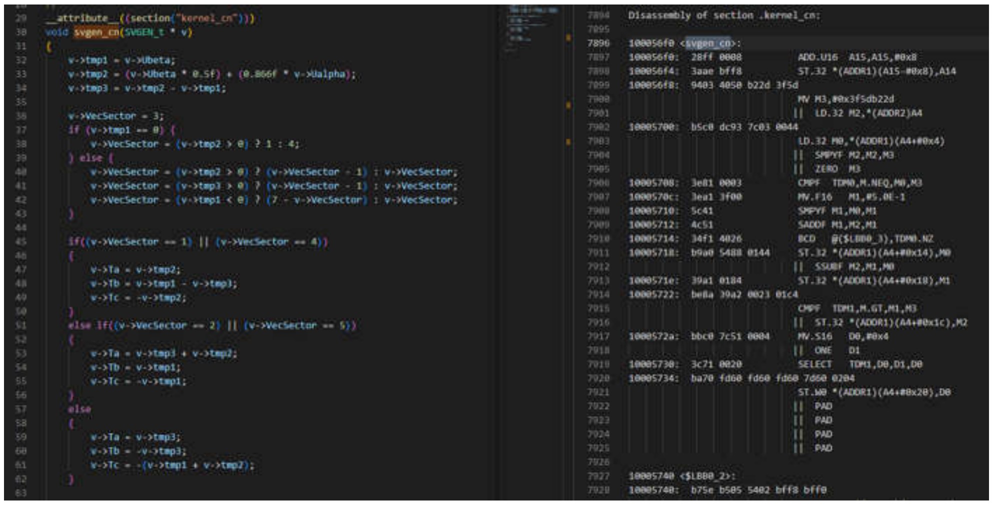
Dead zone code commonly occurs in real-time applications.

The code block below shows the the ternary operator '?' based approach to implementing dead zone code. The C29 (10 cycles, independent of input) outperforms the C28 (25-36 cycles, dependent on input) and Cortex-M7 (23-35 cycles, dependent on input). This efficiency is achieved through the delayed return instruction (RETD) and well-utilized delay slots containing compare (CMPF) and assignment (SELECT) instructions.

```
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 Space Vector Generation (SVGEN) Example

Space Vector Generation (SVGEN) is a common function found in motor control systems, where a vector (α, β) is mapped to a 6-segment space vector, to generate 3 PWM signals. In a normal implementation of SVGEN, as shown in Figure 3-5, if..else statements are used (left-side of Figure), and the compiler generates code that contains branches (right-side of Figure).



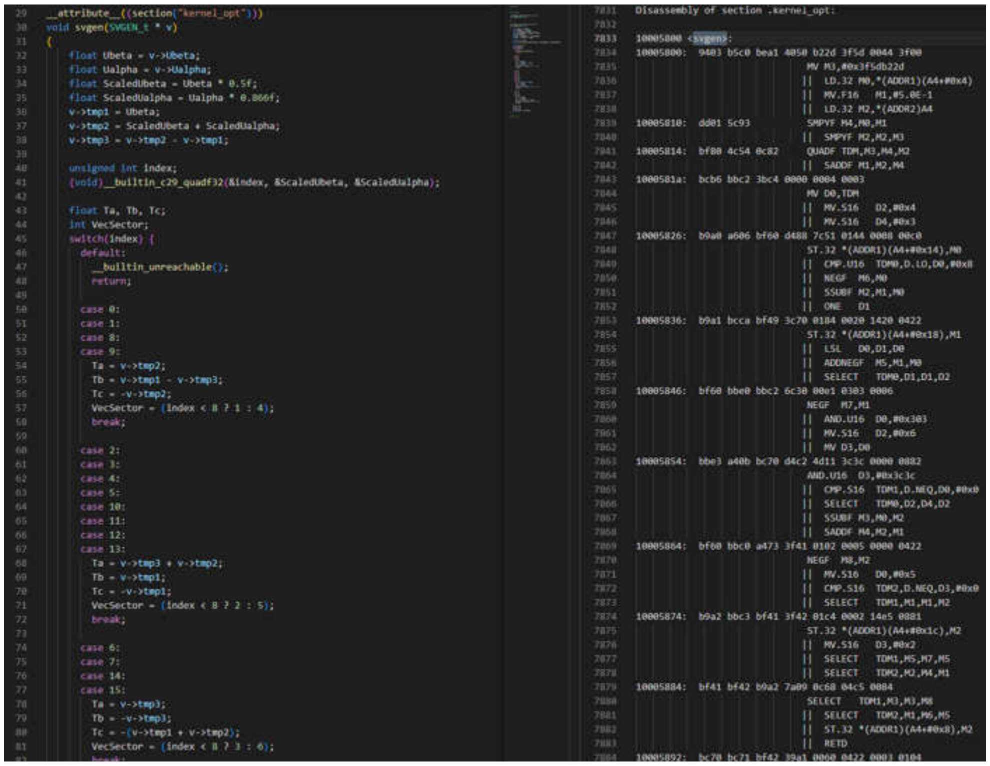
```
__attribute__((section("kernel_cn")))
void svgen_cn(SVGEN_t * v)
{
    v->tmp1 = v->Ubeta;
    v->tmp2 = (v->Ubeta * 0.5f) + (0.866f * v->Ualpha);
    v->tmp3 = v->tmp2 - v->tmp1;

    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.W0 *(ADDR1)(A4+#0x20),D0
                                ||  PAD
                                ||  PAD
                                ||  PAD
                                ||  PAD

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

Figure 3-5. Normal Implementation of SVGEN

In an optimized implementation of SVGEN, as shown in Figure 3-6, the implementation uses the QUADF instruction of the C29, through an intrinsic, __builtin_c29_quadf32. The instruction breaks up the 2-D space into 16 segments. Then a switch() statement maps the 16-segment space to 6-segment space. The C code is illustrated on the left side of the Figure, and the compiler generated assembly on the right side of the Figure. The generated assembly is now straight line code, without branches, and is parallelized (four instructions in parallel every cycle).

Optimized implementation takes 24 cycles on the C29, irrespective of the inputs, whereas the normal implementation takes 26-43 cycles, depending on the inputs. On the C28, the normal implementation takes 70-100 cycles. On the Cortex-M7, the normal implementation takes 58-73 cycles, depending on the inputs.



```
__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);
```

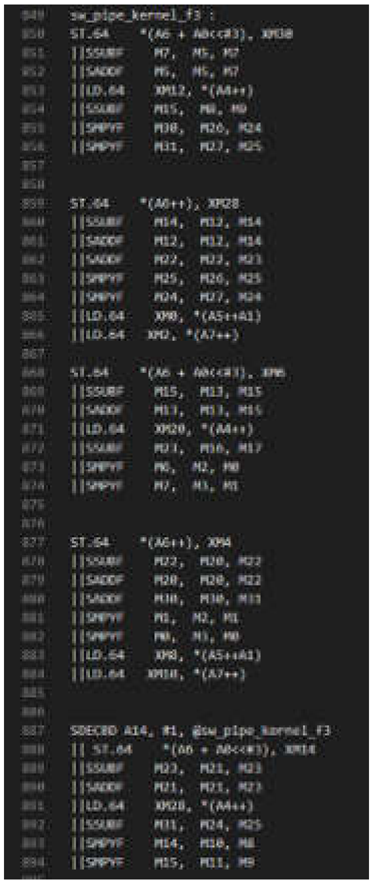
```
Disassembly of section .kernel_opt:

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 0884
                                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
```

Figure 3-6. Optimized Implementation of SVGEN on C29

TI provides libraries covering real-time control and DSP. Specific cases where an optimized implementation of a library yields performance improvements over a natural implementation are called out.

3.2.1.4 Software Pipelining

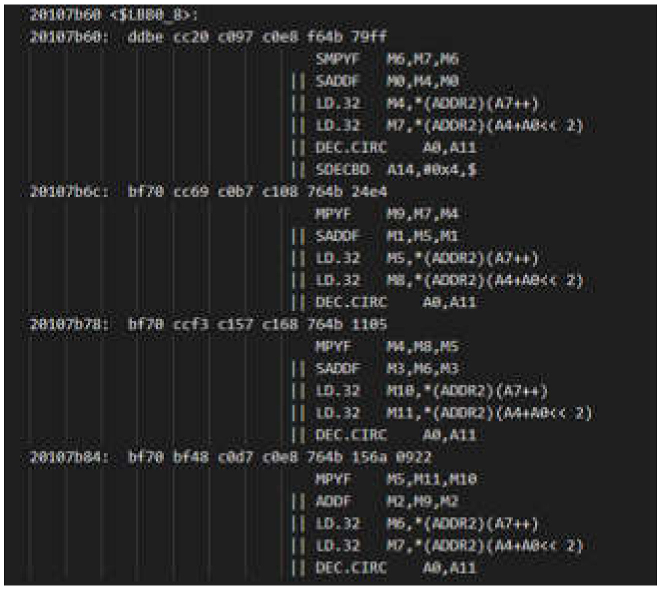
Software pipelining of loops allows multiple iterations of loops to execute in parallel, leveraging the VLIW architecture of the C29 CPU. In Figure 3-7, software pipelining is illustrated for the CFFT. The assembly is hand-written, where the complete 128-bit instruction packet is used and 8 instructions are executed in parallel per cycle in the loop.



```
sw_pipe_kernel_f3 :
ST.64      *(A6 + A0<<#3), XM30
 ||SSUBF     M7,  M5, M7
 ||SADDF     M6,  M5, M7
 ||LD.64     XM12, *(A4++)
 ||SSUBF     M15,  M8, M9
 ||SMPYF     M30,  M26, M24
 ||SMPYF     M31,  M27, M25


ST.64      *(A0++), XM28
 ||SSUBF     M14,  M12, M14
 ||SADDF     M12,  M12, M14
 ||SADDF     M22,  M23, M23
 ||SMPYF     M25,  M26, 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     M30,  M30, M31
 ||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
```

Figure 3-7. Software Pipelining in CFFT - Handwritten Assembly

With -O3 optimization, the C29 compiler generates software pipelined loops, as shown in Figure 3-8, for the FIR. Software pipelining allows loops to perform faster.



```
20107b60 <$LB80_8>:
20107b60:  ddbe cc20 c097 c0e8 f64b 79ff
                                SMPYF   M6,M7,M6
                             || SADDF   M0,M4,M0
                             || LD.32   M4,*(ADDR2)(A7++)
                             || LD.32   M7,*(ADDR2)(A4+A0<< 2)
                             || DEC.CIRC    A0,A11
                             || SDECBD  A14,#0x4,$
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
```

Figure 3-8. Software Pipelining in FIR - Compiler Generated

The compiler generates software pipelined loops at -O3 optimization setting that boosts performance for code with loops.

3.2.2 Customer Control and Math Benchmarks

Figure 3-9 shows C29 performance compared to the C28 CPU *(in cycles)* on select benchmarks that were received directly from customers (denoted A through E). These represent actual customer representative code varying in functionality from Math to Motor Control to Interpolation. C_Motor represents a dual motor benchmark, where two motor instances are run. The parallel C29 architecture is used in this benchmark, resulting in more than five times better performance *(in cycles)* than the C28 CPU.

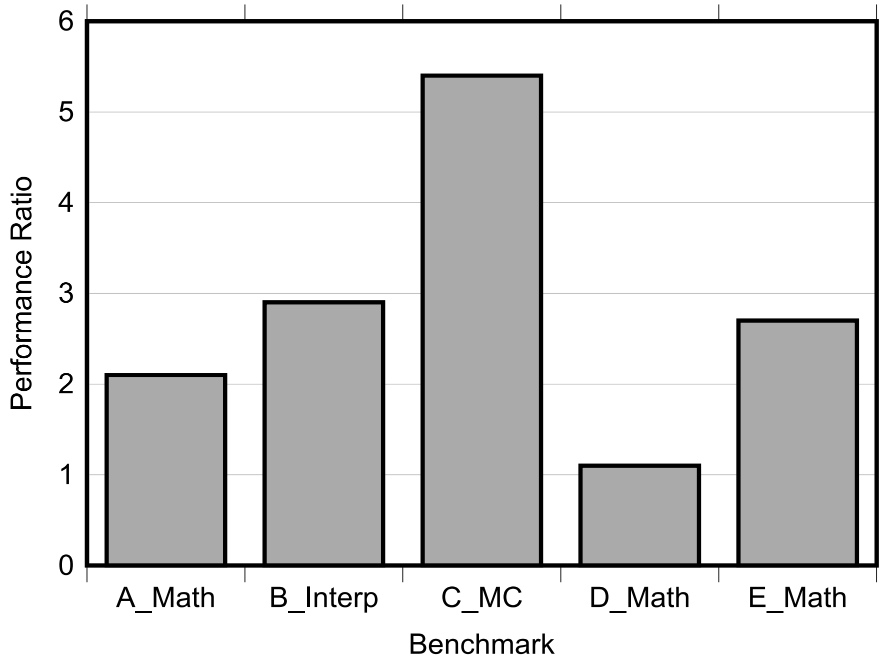


Figure 3-9. Customer Control and Math Benchmarks

Parallel architecture is not used effectively in the D_Math benchmark, which contains only volatile variables. With volatile variables, the compiler performs a load from or a store to memory every time a variable is used. This removes the ability to store the variable in a register and minimize memory accesses until absolutely necessary. Therefore, carefully consider using volatile variables in code.

3.3 General Purpose Processing (GPP) Performance

In addition to real-time control, the C29 CPU has excellent GPP performance. Figure 3-10 shows C29 performance compared to the C28 CPU on select benchmarks that were received directly from customers (denoted F and G). F and G benchmarks represent actual customer benchmarks containing GPP code. F_GPP contains over 100 if() statements and G_GPP contains over 30 if() statements. The benchmarks also contain logical, bit-wise, and arithmetic operations. Figure 3-11 shows C29 versus Cortex-M7 performance for these same benchmarks, and Figure 3-12 shows C29 versus proprietary CPU A performance. Not only is the C29 nearly three times better *(in cycles)* than the C28 CPU at GPP code, it is almost 50% better *(in cycles)* than the Cortex-M7, and twice better *(in cycles)* than proprietary CPU A.

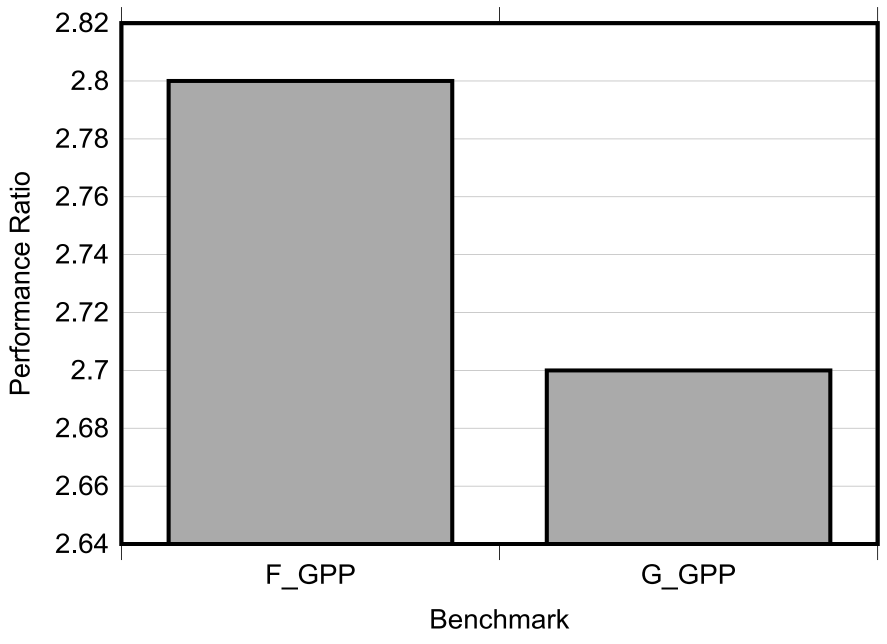


Figure 3-10. C29 versus C28 GPP Performance

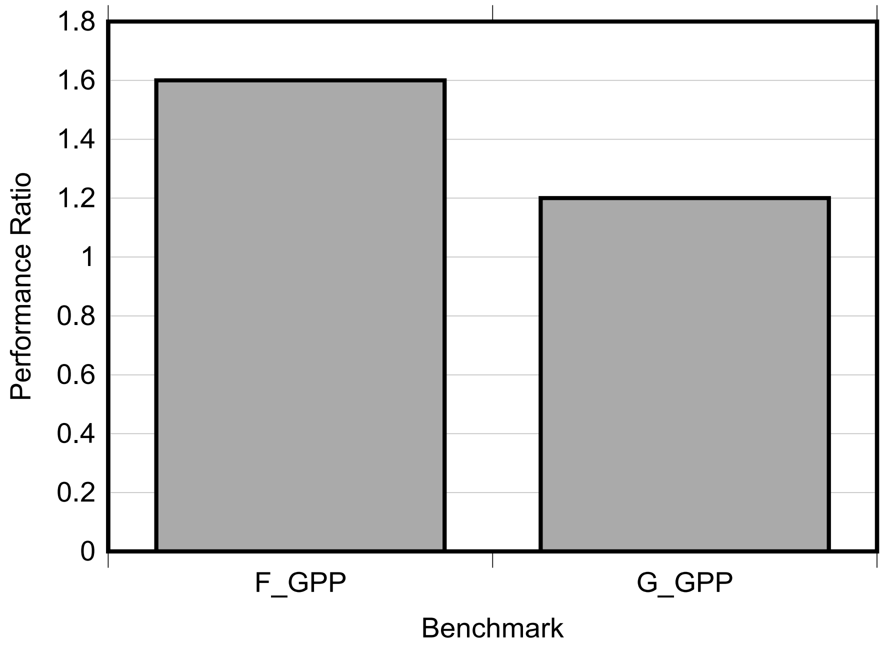


Figure 3-11. C29 versus M7 GPP Performance

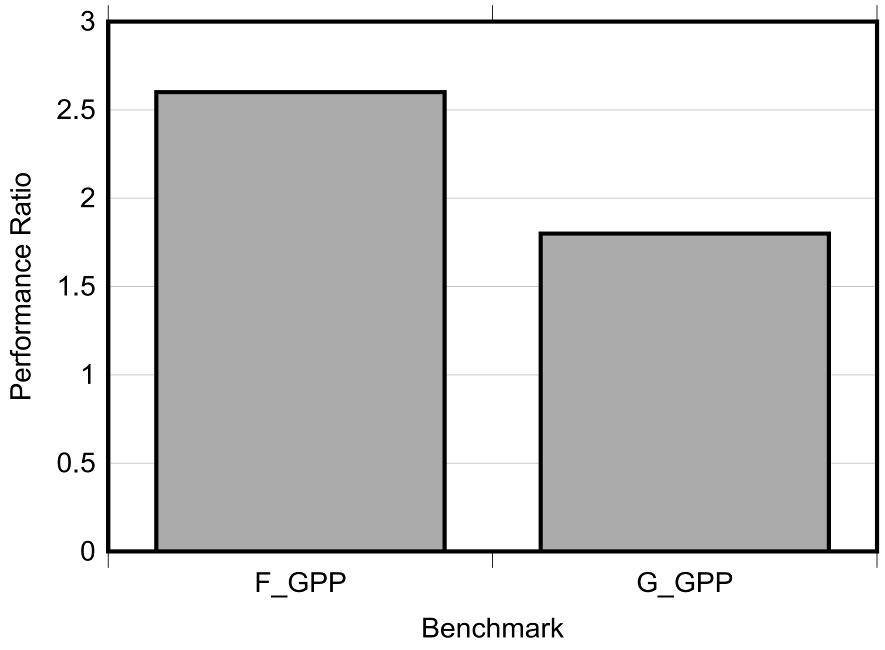


Figure 3-12. C29 versus Proprietary CPU A Performance

3.3.1 Examples and Factors Contributing to Results

This section provides insight and analysis into the architecture and compiler to explain the results illustrated above. The C29 CPU's improved GPP performance is attributed to a number of enhancements:

- Multiple general purpose functional units in the C29 CPU boost general purpose performance.
- Delayed branches, leading to effectively no branch penalties; this is explained in the discontinuity management sub-section.
- Condition execution instructions for short branches; this is illustrated in the saturation and deadzone examples.
- Special branch instructions that allow the C29 compiler to collapse multiple branch destinations into a single instruction, illustrated in the switch example sub-section.

3.3.1.1 Discontinuity Management

Traditionally, Branch, Call, and Return operations incur overhead because of the instruction pipeline. The CPU fetches, decodes, and determines that a branch, call, or return operation needs to occur in the Decode-2 phase of the pipeline. By this time, the pipeline is filled with next instructions, which need to be flushed before the instruction at the discontinuity destination is fetched. Flushing of instructions results in overhead.

The C29 CPU has a 9-stage pipeline, with discontinuity decision occurring in the Decode-2 (D2) phase of the pipeline. Therefore, three instructions following a discontinuity instruction are already in the pipeline (the Fetch-1, Fetch-2, and Decode-1 phases of pipeline). In addition to regular branch, call, or return instructions, the C29 ISA supports *delayed* branch, call, or return instructions (the corresponding instruction has a trailing D, for example CALLD, RETD). When these delayed discontinuity instructions are used, three instructions immediately following them are always executed, regardless of whether the discontinuity occurs or not (in the case of a conditional branch). The three instructions following a delayed discontinuity instruction are referred to as *delay slots*. The C29 Compiler, when using the delay slot version of these instructions, inserts appropriate instructions into delay slots, thus reducing the discontinuity overhead from three cycles to effectively zero cycles.

Two examples illustrating the use of this by a compiler are shown below.

- A function call where 6 function arguments are passed in three delay slots.

```
@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
```

- A return with where the saved registers are restored and the stack is deallocated in three delay slots.

```
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
```

The above examples are models of how the C29 compiler uses delay slots. In practice, delay slots are used for more than just function argument passing and register restoration and stack deallocation. Delay slots often contain instructions for implementing the actual functionality of user code.

3.3.1.2 Switch() Example

Special branch instructions allow the C29 compiler to collapse multiple branch destinations into a single instruction. *switch* is a common construct that occurs in general purpose code, such as housekeeping tasks. The C29 ISA has multiway branch instructions QDECB and DDECB that allow very efficient implementation. In quad decrement branch (QDECB), four destinations are allowed, with the fifth option being linear execution. In dual decrement branch (DDECB), two destinations are allowed, and the third option is linear execution.

A 16 case switch statement is illustrated in the code block below. On the C29 CPU, the switch is implemented with one branch instruction (BCMP) and four QDECB instructions, taking 10 to 17 cycles, depending on the input. On the Cortex-M7, the switch is implemented with compare and branch instructions for each case, thus taking 6 to 51 cycles, depending on the input.

```
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 Model-Based Design Benchmarks

Customers are increasingly shifting towards model-based design and auto code generation. Thus, it is important to understand the performance expected with auto code generation tools, such as Embedded Coder from The Mathworks. At the time of this publication, the C29 is not yet supported in a released version of Embedded Coder, therefore C code generated for the C28 CPU is used for benchmarking. The Sensorless Field Oriented Control based motor control model consists of closed loop control and a Sliding Mode Observer (SMO).
The generated code has real-time control components, as well as GPP components. Model-Based Design Benchmarking shows the benchmarking results, which illustrates the performance of the C29 is more than twice better *(in cycles)* than the Cortex-M4 based competition MCU.

Table 3-2. Model-Based Design Benchmarking

MCU	Cycles	Performance Ratio
#6 (Cortex-M4)	877	1
F29H85x (C29)	393	2.23
F29H85x (C29)	312 (with some hand optimization of generated code)	2.81

3.5 Application Benchmarks

Until now, the presented benchmarks have covered a real-time signal chain, customer benchmarks, specific control and DSP blocks, as well as general purpose benchmarks. This section describes benchmarking results comparing C29 and C28 performance using C2000 reference designs focused on real-time applications such as Digital Power and Motor Control.

C29 based reference designs have not been released, and are a work in progress, therefore early benchmarking results are presented here.

3.5.1 Single Phase 7kW OBC Description

TIDM-2013 is a reference design built using the F28x family of devices to implement single phase OBC design. The design consists of an interleaved continuous conduction mode (CCM) totem-pole (TTPL) bridgeless power-factor correction (PFC) power stage followed by a CLLLC DCDC power stage. The design runs the following ISRs at the mentioned frequencies:

- ISR1 (PWM Update): 120KHz
- ISR2 (PFC Current, CLLC Voltage Loop): 120KHz
- ISR3 (PFC Voltage Loop + Instrumentation): 10KHz

Figure 3-13 shows benchmarks for the above ISRs comparing F29x versus F28x running at the same CPU clock frequency of 200MHz. ISR1 does not change between F29x and F28x because the primary operations in it are PWM peripheral register writes, which do not change between F28x and F29x. ISR2 on F29x is 1.7 times faster compared to F28x.

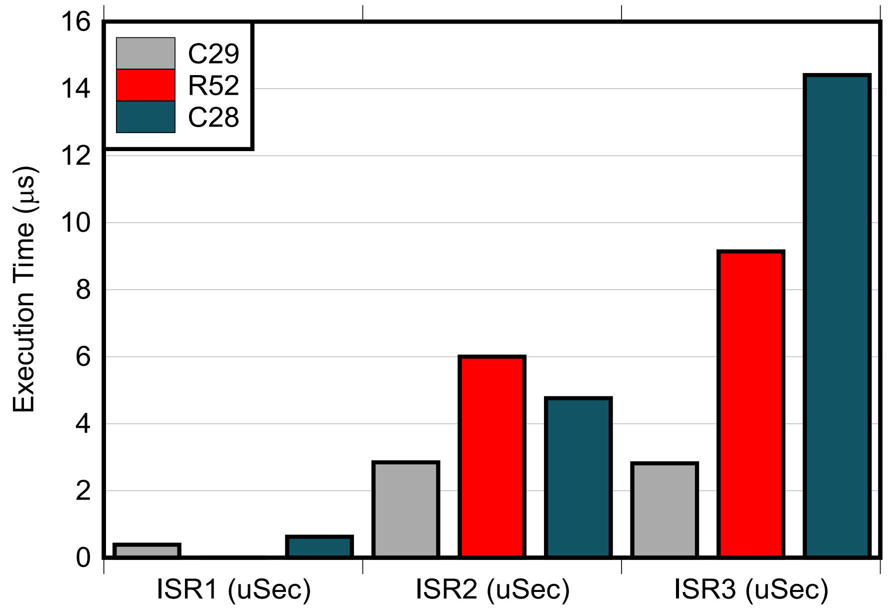


Figure 3-13. OBC Benchmark

3.5.2 Vienna Rectifier-Based Three Phase Power Factor Correction

Vienna rectifier power topology is used in high power three phase power factor (AC-DC) applications, such as off board EV chargers and telecom rectifiers. TIDM-1000 illustrates a method to control the power stage using C2000™ microcontrollers (MCUs). TIDM-1000 Benchmarking shows early benchmarking results on Lab four (closed voltage loop with inner current loop and midpoint voltage balancing). The cycles are measured inside the ISR from start to finish. The results show the F29x achieves twice *(in cycles)* the performance of the C28 CPU.

Table 3-3. TIDM-1000 Benchmarking

TIDM-1000	Cycles	Performance Ratio
F2837x (C28)	308	1
F29H85x (C29)	153	2.01

3.5.3 Single-Phase Inverter

TIDM-HV-1PH-DCAC implements single-phase inverter (DC-AC) control using the C2000™ F2837xD and F28004x microcontrollers. TIDM-HV-1PH-DCAC Benchmarking shows early benchmarking results on Lab 3 (closed voltage loop with inner current loop). The cycles are measured inside the ISR from the beginning until the end. The results show the C29 achieves 80% better performance *(in cycles)* than the C28 CPU.

Table 3-4. TIDM-HV-1PH-DCAC Benchmarking

TIDM-HV-1PH-DCAC	Cycles	Performance Ratio
F2837x (C28)	609	1
F29H85x (C29)	332	1.83

3.5.4 Machine Learning

Machine Learning (ML) techniques in real-time control are emerging, with applications such as arc fault detection and motor fault detection. Artifical Intelligence (AI) accelerators on-chip are becoming common to run embedded AI models. However, ML performance on real-time control CPUs is also an important consideration. Machine Learning Benchmarks shows benchmarks of 3, 4, and 5-layer Computational Neural Networks (CNN) on a Cortex-M7 MCU and the C29 based F29H85x. The C29 is almost five times faster than the Cortex-M7, even with the latter operating at twice the CPU frequency.

Table 3-5. Machine Learning Benchmarks

Model	Cortex-M7 400MHz, floating-point model (milliseconds)	F29H85x (C29) 200MHz, floating-point model (milliseconds)
3-layer CNN	11.54	2.33
4-layer CNN	11.82	2.35
5-layer CNN	12.02	2.30

3.6 Flash Memory Efficiency

Flash execution efficiency is important because not all code can run from zero wait-state memory. On F29H85x at 200MHz, three wait-states are needed for flash access. Further, pre-fetch and block cache mechanisms are available to mitigate the effect of the wait states, and they are enabled. Flash Efficiency Benchmarks shows flash efficiency results in percentages (%) for some benchmarks comparing F29H85x and F2837x (also three wait-states at 200MHz). For many benchmarks, running from flash is akin to running from zero wait-state memory.

Table 3-6. Flash Efficiency Benchmarks

Benchmark	F2837x (%)	F29H85x (%)
CFFT	81	93
FIR	91	86
IIR (loop)	82	99
Signal-chain (ACI)	95	97
Binary LUT search	82	97

3.7 Code-size Efficiency

In addition to performance efficiency, code-size efficiency is an important metric, especially when zero wait-state memory is limited. Performance critical code is usually run out of zero wait-state memory, and non performance critical code is run from Flash memory. Code-size Benchmarks shows code-size efficiency of various benchmarks, comparing C29 with C28 and ARM (Cortex-M7). A few points are noted from the results:

- C29 code-size is mostly comparable to C28, as well as Cortex-M7 code-size. In some benchmarks, the C29 achieves lower code-size, and in some other benchmarks higher code-size.
- Code-size results correspond to the -O3 optimization setting of the compiler. The user has the flexibility to selectively use -Oz on portions of code to reduce code-size.
- C29 FIR code-size is larger because of software pipelining (which results in a huge performance boost). Loops, in general, are a small part of overall code.

Table 3-7. Code-size Benchmarks

DSP, Math, and Real-time Benchmarks	C28 versus C29 code-size (C) <1 implies C28 code-size is smaller	Cortex-M7 vs C29 code-size (C) <1 implies Cortex-M7 code-size is smaller
FIR	0.5	0.7
IIR	0.7	1.4
DCL (Digital Control Library)	1.5	1.5
FCL (Fast Current Loop)	1.1	1.1
SPLL (Software Phase Locked Loop)	1.7	1.2
SVGEN	1.2	1
ACI signal chain	0.9	0.6
Customer DSP, Math, and Real-time Benchmarks		
B_Interp	1.1	1
C_Motor	1.3	1
D_Math	0.8	0.8
E_Math	1	0.7
GPP Benchmarks		
F_GPP	0.8	0.8
G_GPP	0.7	0.7
Reference designs		
Vienna Rectifier	0.94	–
Single-phase inverter	0.75	–

4 Summary

Industrial and Automotive applications requirements on efficiency and power density are ever increasing. Thus, the need has never been greater for scalable real-time MCUs with enhanced performance, that enable advanced topologies and integration options, and offer inbuilt safety and security features. The new C29 CPU with unrivaled real-time performance is highly optimized to meet the above challenges. The C29 CPU's parallel architecture enables implementing in a single core what traditionally requires multiple CPUs. This white paper demonstrates a broad spectrum of benchmarks that affirm the C29 CPU's capability. The C29 compiler provides out of the box performance entitlement on C code. The SSU tightly coupled to the C29 CPU allows users to seamlessly develop secure, ASIL-D safety applications, without the need for reprogramming.

5 References

1. Texas Instruments, F29H85x and F29P58x Real-Time Microcontrollers data sheet
2. Texas Instruments, F29H85x and F29P58x Real-Time Microcontrollers technical reference manual
3. Texas Instruments, Application Software Migration to the C29 CPU user's guide
4. Texas Instruments, Implementing Run-Time Safety and Security Protections With the C29x SSU application note
5. Texas Instruments, TI C29x Clang Compiler Tools user's guide
6. Texas Instruments, Real-time Benchmarks Showcasing C2000 Control MCU's Optimized Signal Chain application note
7. Texas Instruments, TMS320F2837x, TMS320F2838x, TMS320F28P65x Migration to TMS320F29H85x