Application Note

Real-time Benchmarks Showcasing C2000™ Control MCU's Optimized Signal Chain

Ashwini Athalye

ABSTRACT

The key to real-time control is reduced latency of performing sensing, processing and actuation defined together as the real-time signal chain. Many software benchmarks only focus on the processing aspect typically expressed in million instructions per second (MIPS), without full regard for the interaction between peripherals, CPU, and co-processors. Such benchmarks do not provide a full view of the real-time performance capabilities of a system. This application note describes a real-time benchmark created around a real world control application that highlights the intricacies of real-time control and the need for this more comprehensive benchmarking approach. The data from the real-time benchmark provides an insight into features of the C2000™ control MCUs that make it an excellent platform for real-time control applications.

Table of Contents

List of Figures

List of Tables

Trademarks

C2000™ and Code Composer Studio™ are trademarks of Texas Instruments.
All trademarks are the property of their respective owners.

1 Introduction

A real-time benchmark gives a holistic view of signal chain performance. As illustrated in Figure 1-1, this includes interrupt response time, context save overhead, peripheral reads and writes along with control algorithms, all of which together compose a real-time control operation.

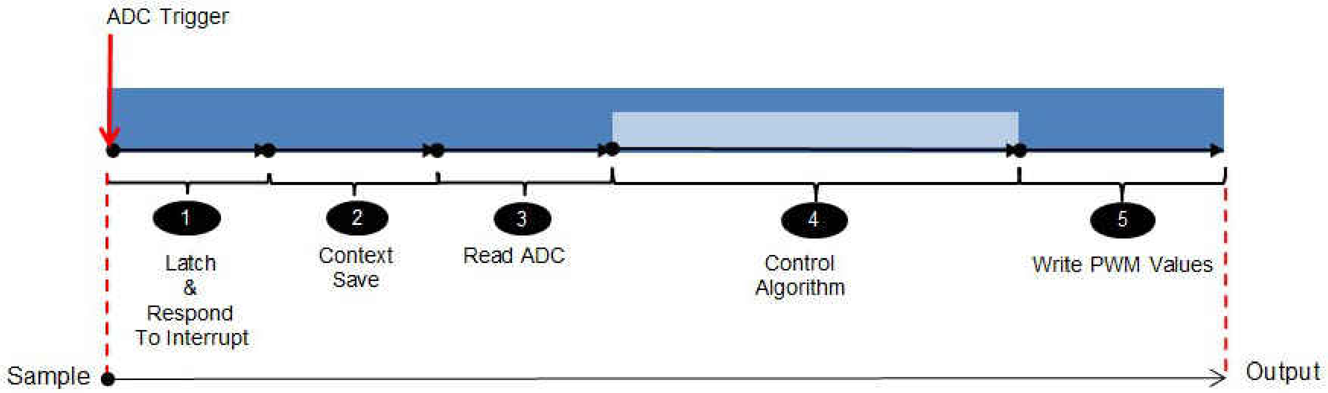


Figure 1-1. Real-Time Signal Chain

The C2000 architecture has unique capabilities that make it a microcontroller optimized for real-time control by minimizing the sample to output time. The benchmarking in this document demonstrates these C2000 features.

2 ACI Motor Control Benchmark Application

The ACI Motor Control Benchmark simulates the sensorless AC induction motor control application. The application performs all the typical operations: analog-to-digital converter (ADC) reads for sensing phase currents, transform blocks that operate on the sensed current and PWM writes to control phase voltages. No special external hardware is needed to provide stimulus as the application has a block of code that models the behavior of an induction motor. To simulate closed loop behavior, the expected current from the motor model is fed into the ADC via the DAC modules. A single ADC is configured to sense the phase A and phase B currents via two channels sequentially. Phase C current is not sensed as it can be derived from phase A and phase B currents. Three PWM writes simulate control of duty cycle of the three phase A, B and C voltages. Figure 2-1 represents the execution blocks in the control loop interrupt routine of the benchmark application. The control loop interrupt is triggered at a rate of 2 KHz and 1024 iterations of the control loop interrupt routine are executed before the application terminates.

In this block diagram, the dark gray blocks represent the peripheral accesses, the light gray blocks represent the control algorithms and the white blocks represent the 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.

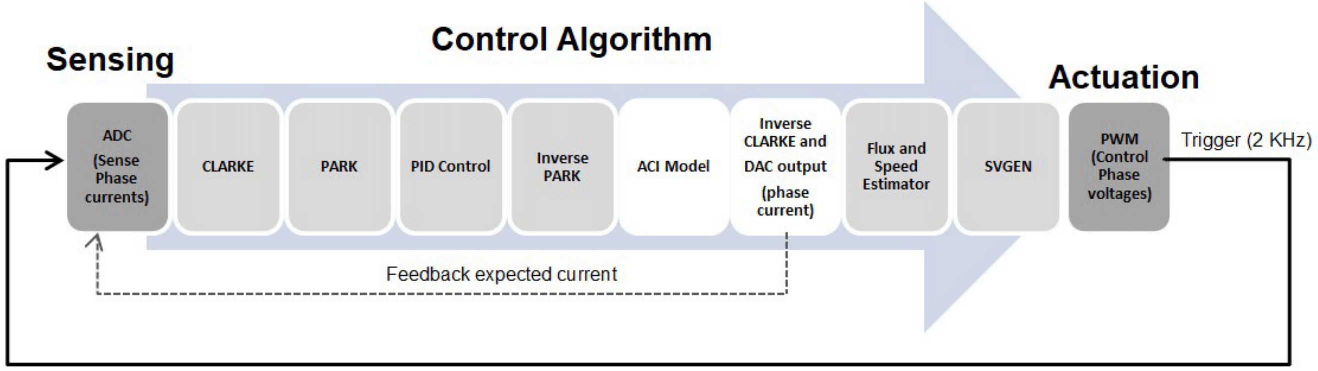

Sensing
Control Algorithm
Actuation
ADC (Sense Phase currents)
CLARKE
PARK
PID Control
Inverse PARK
ACI Model
Inverse CLARKE and DAC output (phase current)
Flux and Speed Estimator
SVGEN
PWM (Control Phase voltages)
Trigger (2 KHz)
Feedback expected current

Figure 2-1. ACI Motor Benchmark Application Block Diagram

2.1 Source Code

The software benchmark example is available in C2000Ware.

- **C28x** CPU benchmark available in C2000Ware version **3.04.00.00** onwards
- **CLA** accelerator benchmark available in C2000Ware version **4.00.00.00** onwards.

The example can be found at the following location within the C2000Ware installation.

<C2000Ware>\examples\demos\benchmark\aci_motor_benchmark

The top level folders are:

- common
- device_support
- f28004x
- f2837x

The source code is located in the '*common'* and '*device_support'* folders. The '*common*' folder contains device independent code such as transform algorithms, motor modeling code, and so forth. The *'device_support'* folder contains code that is specific to the particular device such as device configuration, peripheral read/write. and so forth within sub folders named for that device.

The application is supported in Code Composer Studio™ (CCS) for TMS320F28004x and TMS320F2837x devices and can be executed on TMS320F28004x Launchpad as well as TMS320F2837x Launchpad. The respective CCS projects are located in the sub-folder *'ccs'* within the *'f28004x'* and '*f2837x'* top level folders.

The application has multiple implementation variants; one variant uses the math engine Trigonometric Math Unit (TMU) for performing trigonometric calculations needed by the Park, Inverse Park and Flux Estimator control algorithms, the other variant uses a software library called FastRTS to perform the trigonometric calculations. The FastRTS library is included in C2000Ware, the library and documentation can be found at <C2000Ware>\libraries\math\FPUfastRTS\c28\. The goal with these two variants is to show the performance boost the math engine TMU can provide as compared to a software library based implementation.

Another set of implementation variants involve the CLA. One variant is with the C28x CPU offloading part of the compute to CLA and another variant is with the benchmarking control code executing entirely from CLA only The goal of these two variants is to show how CLA can be used to aid in meeting real-time goals.

2.2 CCS Project for TMS320F28004x

The project can be imported in CCS using "Import Project" option and selecting the project located at '*f28004x\ccs*'. The project comes with several pre-defined build configurations. Once the project is imported, the project can be built for these different build configurations that are described below:

- ***SignalChain_RAM_TMU:***The application compiles for the full signal chain executing out of RAM using the TMU instructions for control algorithms like Park and Inverse Park that use trigonometric math. The build setting is controlled by the defines SIGNAL_CHAIN=1 and USE_FAST_TRIG_LIB=0.
- ***SignalChain_FLASH_TMU:*** The application compiles for the full signal chain similar to the SignalChain_RAM_TMU except that it executes out of FLASH. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=0 and _FLASH.
- ***SignalChain_RAM_FastRTS:*** The application compiles for the full signal chain executing out of RAM using the FastRTS library instead of TMU for control algorithms like Park, Inverse Park that use trigonometric math. The build setting is controlled by the defines SIGNAL_CHAIN=1 and USE_FAST_TRIG_LIB=1.
- ***SignalChain_FLASH_FastRTS:*** The application compiles for the full signal chain similar to the SignalChain_RAM_FastRTS except that it executes out of Flash. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=1 and _FLASH.
- ***ControlAlgo_RAM_TMU:*** The application compiles for just the control algorithm and not the full signal chain, that is, peripherals are not configured and interrupts are not generated in this build configuration. This can be used as a starting point for porting this application to other devices. The build setting is controlled by the defines SIGNAL_CHAIN=0 and USE_FAST_TRIG_LIB=0.

- ***SignalChain_RAM_TMU_CLA_OFFLOAD:*** The application compiles for the full signal chain executing out of RAM using the TMU instructions for control algorithms like Park and Inverse Park that use trigonometric math on the C28x CPU side. The C28x CPU offloads some of the control code compute to CLA resulting in parallelized execution. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=0 and CLA_OFFLOAD.
- ***SignalChain_FLASH_TMU_CLA_OFFLOAD:*** The application compiles for the full signal chain similar to the SignalChain_RAM_TMU except that the C28x CPU code executes out of FLASH. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=0, CLA_OFFLOAD and _FLASH.
- ***SignalChain_RAM_CLAmath_CLA:*** The application compiles for the full signal chain executing out of RAM from CLA using the CLAmath library for control algorithms like Park and Inverse Park that use trigonometric math as unlike C28x CPU, the CLA accelerator does not have a TMU. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=0 and CLA_CPU.

Once the application is built, load the application to the target and select the *run* option in CCS to execute the application.

The data from the seven SignalChain build configurations are used in this application report to compare TMU vs FastRTS execution, RAM vs Flash execution and CLA execution.

2.3 CCS Project for TMS320F2837x

The project can be imported in CCS using "Import Project" option and selecting the project located at '*f287x\ccs*'. The project comes with several pre-defined build configurations. Once the project is imported, the project can be built for different build configurations which are described below:

- ***SignalChain_RAM_TMU:*** The application compiles for the full signal chain executing out of RAM using the TMU instructions for control algorithms like Park, Inverse Park that use trigonometric math. The build setting is controlled by the defines SIGNAL_CHAIN=1 and USE_FAST_TRIG_LIB=0.
- ***SignalChain_FLASH_TMU:*** The application compiles for the full signal chain similar to the SignalChain_RAM_TMU except that it executes out of Flash. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=0 and _FLASH.
- ***SignalChain_RAM_TMU_CLA_OFFLOAD:*** The application compiles for the full signal chain executing out of RAM using the TMU instructions for control algorithms like Park and Inverse Park that use trigonometric math on the C28x CPU side. The C28x CPU offloads some of the control code compute to CLA resulting in parallelized execution. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=0 and CLA_OFFLOAD.
- ***SignalChain_FLASH_TMU_CLA_OFFLOAD:*** The application compiles for the full signal chain similar to the SignalChain_RAM_TMU except that the C28x CPU code executes out of FLASH. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=0, CLA_OFFLOAD and _FLASH.
- ***SignalChain_RAM_CLAmath_CLA:*** The application compiles for the full signal chain executing out of RAM from CLA using the CLAmath library for control algorithms like Park and Inverse Park that use trigonometric math as unlike C28x CPU, the CLA accelerator does not have a TMU. The build setting is controlled by the defines SIGNAL_CHAIN=1, USE_FAST_TRIG_LIB=0 and CLA_CPU.

Once the application is built, go into CCS menu option '*Tools->Debugger Options->Auto Run and Launch Options*' and uncheck '*On program load or restart*' option for '*Auto run to main symbol*'. This is needed because there are limited breakpoints on this device and the printf messages in the application use CIO that requires the use of the break point resources. So, the Auto run to main option needs to be disabled for the application to work correctly. Load the application and select the run option to execute the application.

The data from the five build configurations are used in this application report to compare RAM vs Flash execution as F2837x has a different and better performing Flash than F28004x and to understand CLA execution.

2.4 Validate Application Behavior

The application executes the control loop for 1024 iterations before terminating. The application saves the alpha phase current as well as the per unit motor speed for each execution iteration in buffers DLogCh1 and DLogCh2. These saved values can be plotted in the graphical view in CCS. The profiles of these plots are similar to what would be generated in a real sensorless AC induction motor application. To check that the application has executed correctly, halt the target after the test completes and then graph the saved values by opening '*Tools->Graph->Dual Time*', click on '*Import*' and select the graph properties file '*aci.graphProp*' located in top level folder '*f28004x*'. The same graph properties file can also be used for any of the build configuration applications executing on TMS320F28004x or TMS32F2337x target. The alpha phase current is plotted in DualTimeA graph and the per unit motor speed in DualTimeB graph. The X axis has a data point for each of the 1024 samples and the Y axis has the respective values for each of the data points. The profile of the graphs should be similar to Figure 2-2 with the alpha current taking on a sinusoidal-like pattern and the motor speed converging to the expected value of 0.5 units.

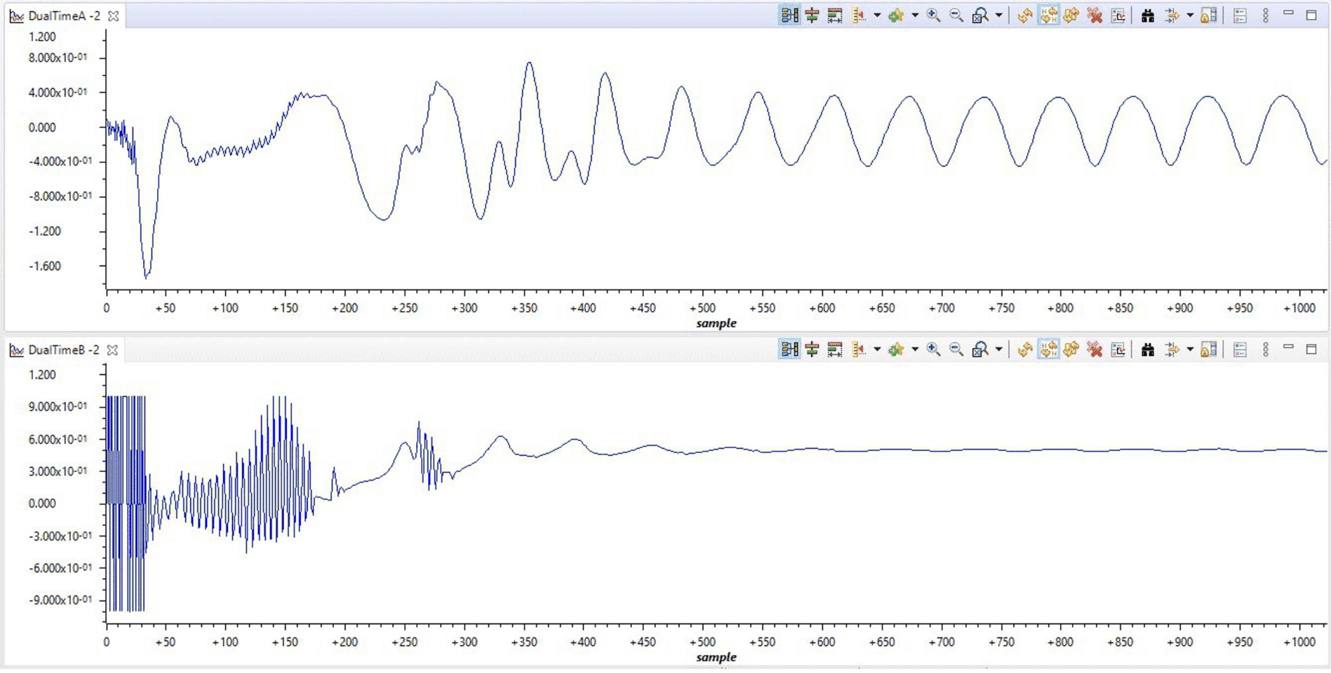
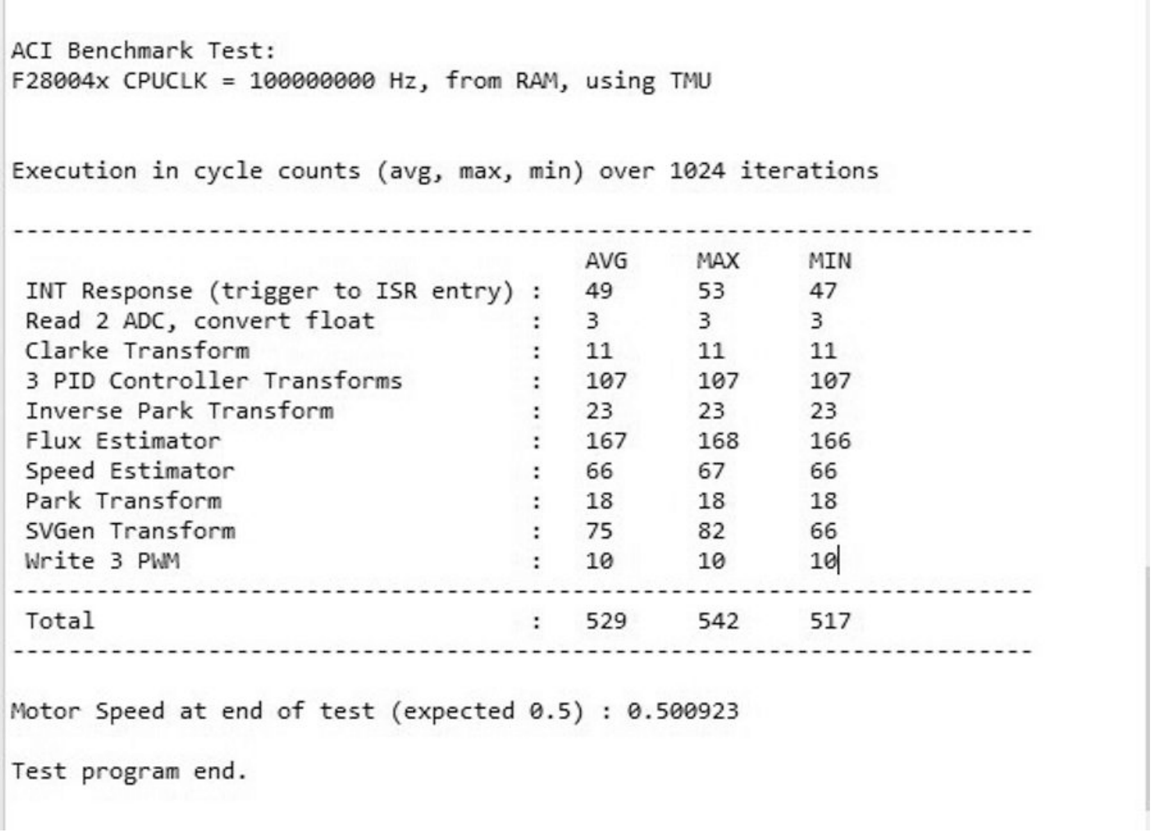


Figure 2-2. Graphical View of Alpha Phase Current (DualTimeA) and Per Unit Motor Speed (DualTimeB)

2.5 Benchmarking Methodology

The application uses a timer counter and a PWM counter to benchmark various parts of the application in cycle counts. As such, no external hardware like a scope is needed for measuring the benchmark results. The benchmark data is collected for each of the 1024 control loop execution iterations and output on the console window by printf messages in the application. The device type, CPU clock frequency and RAM or Flash (with wait state) execution information is output along with cycle counts (avg, max, min) for each of the application execution blocks. Figure 2-3 shows a sample output of execution on F28004x from RAM with TMU.



```
ACI Benchmark Test:
F28004x CPUCLK = 100000000 Hz, from RAM, using TMU


Execution in cycle counts (avg, max, min) over 1024 iterations

-------------------------------------------------------------------------
                                         AVG     MAX     MIN
 INT Response (trigger to ISR entry) :   49      53      47
 Read 2 ADC, convert float           :   3       3       3
 Clarke Transform                    :   11      11      11
 3 PID Controller Transforms         :   107     107     107
 Inverse Park Transform              :   23      23      23
 Flux Estimator                      :   167     168     166
 Speed Estimator                     :   66      67      66
 Park Transform                      :   18      18      18
 SVGen Transform                     :   75      82      66
 Write 3 PWM                         :   10      10      10
-------------------------------------------------------------------------
 Total                               :   529     542     517
-------------------------------------------------------------------------

Motor Speed at end of test (expected 0.5) : 0.500923

Test program end.
```

Figure 2-3. ACI Motor Benchmarking Output

2.5.1 Details of Benchmarking With Counters

Real-time benchmarking requires measuring performance along different phases of the signal chain. As indicated in Figure 2-4, this can be largely broken up into three parts: hardware response, compiler generated context save and user code inside the interrupt service routine(ISR).

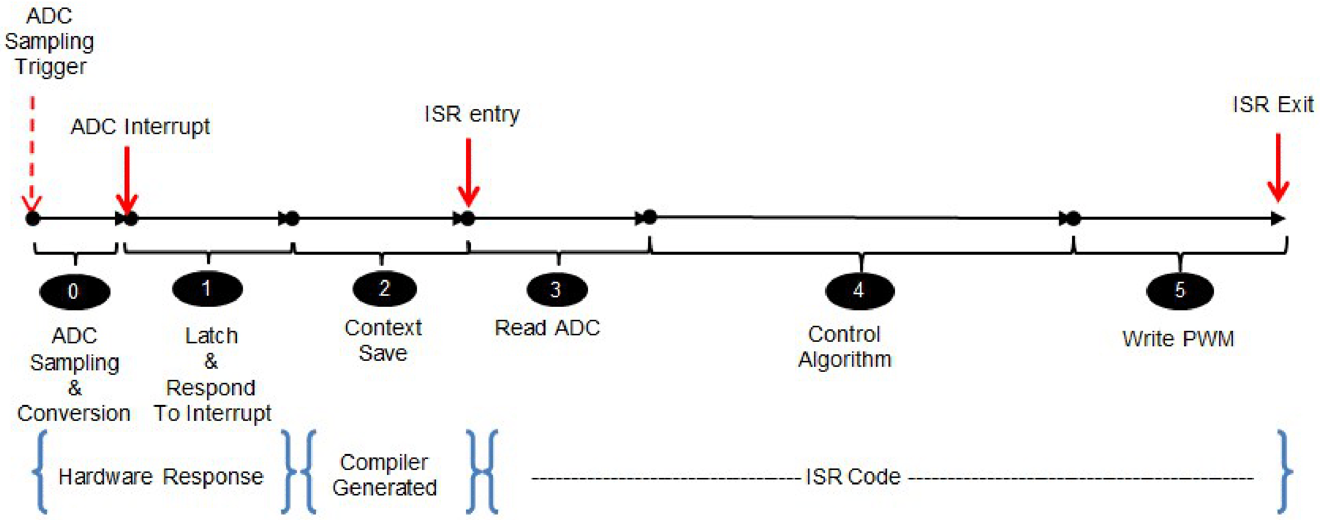


Figure 2-4. Real-Time Benchmarking Phases

Measuring execution inside the ISR is a straightforward task. The timer peripheral can be used to benchmark code inside the ISR by inserting pieces of code that read the timer counter before and after a block of code and calculating the difference. The application measures the control code and the ADC read and PWM writes using this method. This approach however is not possible for the compiler generated context save and for the hardware response measurement.

In this particular application, the PWM is used to trigger ADC sampling when the PWM counter reaches a certain value (timebase period) and the ADC in turn generates an interrupt when sampling completes. While it is not possible to measure the hardware response and the compiler generated context save separately, they can be measured together by reading the PWM counter as the first operation inside the ISR code and calculating the difference between the PWM value read and the PWM value when ADC sampling was triggered. This is the method by which the application measures the INT response (hardware response + compiler context save).

2.6 ERAD Module for Profiling Application

The F28004x devices have a hardware module called ERAD (Embedded Real-time Analysis and Diagnostics) which has comparators and counters that can be used for profiling applications. The '*f28004x*' directory contains a script called '*aci_stats_using_erad.js*' that demonstrates how the ERAD can be used non-intrusively and without modifying the application to measure and analyze application execution. The script can be used to verify the number of times the interrupt was taken (should be 1024) and also serves as an example of how ERAD can be used to measure execution cycles from ADC interrupt generation to ISR entry, and ISR user code execution. This script is only provided for ERAD demonstration and application validation purpose. Measurements generated by this script are not used in the real-time benchmarking analysis in this document.

Note

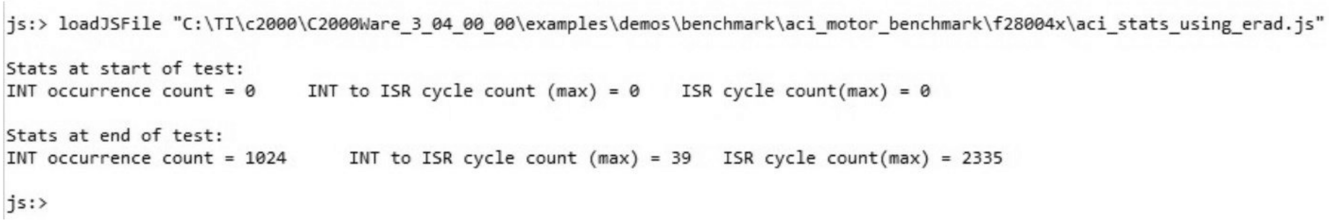
The script only works as is for the F28004x SignalChain_RAM_TMU configuration. The script needs to be modified to benchmark an application built for another configuration as the ISR addresses profiled in the script are hard coded for the "SignalChain_RAM_TMU" configuration build.

Follow these steps to use the script:

1. Load the application compiled for *'SignalChain_RAM_TMU'* configuration to the target.
2. Open Scripting Console view in CCS.
3. Start the script by typing the following command:

   loadJSFile
   ```
   "C:\TI\c2000\C2000Ware_3_04_00_00\examples\demos\benchmark\aci_motor_benchmark\f28004x\aci_stats_using_erad.js"
   ```
4. The script will print out the initial measurement that is 0 for all.
5. Run the target and once the test completes halt the target.
6. Upon halting the target the script will output the captured measurements.

Figure 2-5 shows the output from the script.



```
js:> loadJSFile "C:\TI\c2000\C2000Ware_3_04_00_00\examples\demos\benchmark\aci_motor_benchmark\f28004x\aci_stats_using_erad.js"

Stats at start of test:
INT occurrence count = 0     INT to ISR cycle count (max) = 0    ISR cycle count(max) = 0

Stats at end of test:
INT occurrence count = 1024      INT to ISR cycle count (max) = 39   ISR cycle count(max) = 2335

js:>
```

Figure 2-5. Script Output With Application Stats Profiled by ERAD

Note

The ERAD module measures the "INT to ISR cycle count" that cycles from ADC interrupt generation to ISR entry. This does not account for the ADC sampling and ADC trigger times that are included in the "INT response (trigger to ISR) cycle count" measured by the application code. Hence, the "INT to ISR cycle count" measured by ERAD script is less than the "INT response (trigger to ISR) cycle count" measured by the application code by ~10 cycles, which is equal to the ADC sampling and trigger duration.

3 Real-time Benchmark Data Analysis

As described in previous sections, the application has code that profiles different parts of the signal chain and outputs the measured execution in cycle counts to the console. An analysis of these numbers will reveal features of the C2000 architecture that make it optimized for real-time control. The analysis will also reveal the importance of a real-time benchmarking approach to truly assess a system for its real-time performance.

Note

The benchmarking data for TI C2000 devices presented in this chapter was gathered using application released in C2000Ware v3.04.00.00 in 2021 and executed on TMS320F280049C Launchpad and TMS320F28379D Launchpad. The application was executed using Code Composer Studio v10.1.0.00010 and compiled with TI v20.2.1.LTS compiler with optimization set to the recommended level - 2 (Global optimizations) and optimized for speed - 5. The application was compiled in COFF format.

3.1 ADC Interrupt Response Latency

Interrupt latency is an important factor in understanding the response time of a real-time system. The typical method by which interrupt latency of a system is assessed is the number of cycles it takes for the hardware to respond to an interrupt and branch to the interrupt vector (hardware latch and respond). However, in a real-time application this is only part of the response latency.

Consider the sensorless ACI motor benchmark application, the PWM triggers the ADC to initiate sampling. The ADC takes a certain number of cycles to finish sampling or conversion, which is dependent on the system configuration as well as ADC capability. Once the ADC is ready it generates an interrupt. In the ACI motor benchmark the ADC is configured for early interrupt generation where the interrupt is generated after sampling completes and just as conversion starts instead of at the end of conversion. Once the ADC generates the interrupt, the hardware responds to the interrupt and branches to the interrupt vector. However, the interrupt service routine does not start executing user code right away but before that can happen some compiler generated context save must be performed and then the user code is executed.

In effect, as illustrated in the Figure 3-1, the real interrupt latency is an aggregate of four components: trigger delay + ADC sampling duration + hardware latch and respond + compiler generated context save.

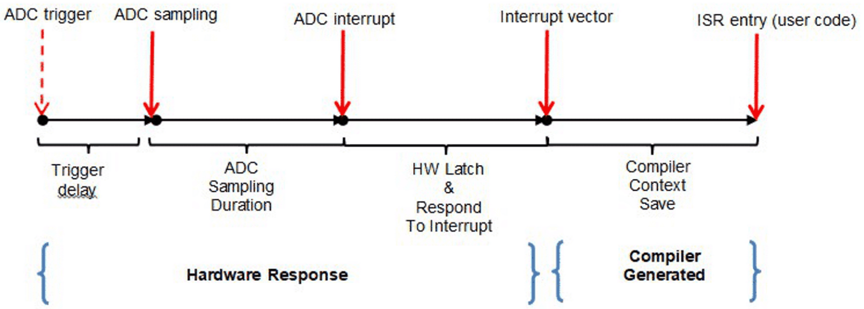
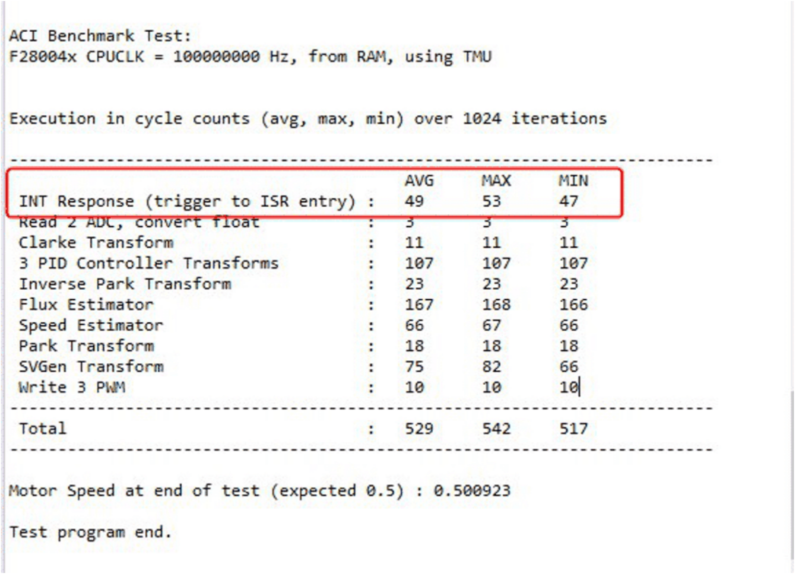


Figure 3-1. ACI Motor Benchmark Interrupt Response Components

The ACI real-time benchmark outputs this value as "INT Response (trigger to ISR entry)". As can be seen in Figure 3-2, the C28x INT response time on an average is 49 cycles and represents the real latency for when an application can start responding to a sensing event for the sensorless ACI motor example. Traditional benchmarks do not measure the system in this fashion. The variance in the INT response cycles between the Min and Max value is expected as there may be additional cycles needed to complete an ongoing background instruction before the asynchronous interrupt event can be taken by the CPU.



```
ACI Benchmark Test:
F28004x CPUCLK = 100000000 Hz, from RAM, using TMU


Execution in cycle counts (avg, max, min) over 1024 iterations

------------------------------------------------------------------------
                                       AVG     MAX     MIN
 INT Response (trigger to ISR entry) :  49      53      47
 Read 2 ADC, convert float           :  3       3       3
 Clarke Transform                    :  11      11      11
 3 PID Controller Transforms         :  107     107     107
 Inverse Park Transform              :  23      23      23
 Flux Estimator                      :  167     168     166
 Speed Estimator                     :  66      67      66
 Park Transform                      :  18      18      18
 SVGen Transform                     :  75      82      66
 Write 3 PWM                         :  10      10      10
------------------------------------------------------------------------
 Total                               :  529     542     517
------------------------------------------------------------------------

Motor Speed at end of test (expected 0.5) : 0.500923

Test program end.
```

Figure 3-2. ACI Motor Benchmark Interrupt Response Benchmark Data

Table 3-1 lists a theoretical estimate of the contribution of each individual element and as can be seen the total cycles aligns with the minimum cycles benchmarked by the application.

Table 3-1. Theoretical Estimate of F28004x ADC INT Response Duration

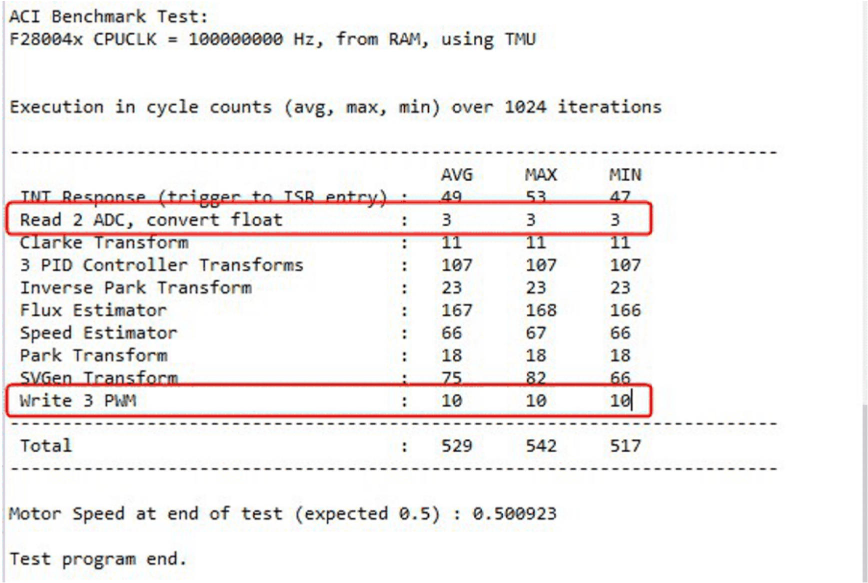
Trigger Delay [1]	ADC Sampling Duration [2]	HW Latch and Response[3]	Compiler Context Save[4]	Total (cycles)
2	8	14	23	**47**

[1] From ADC timing diagram in device-specific data sheet.
[2] Configured by application per device-specific data sheet recommendation.
[3] From peripheral interrupt description in device-specific TRM.
[4] Derived from inspecting generated code and calculating cycles.

A look at the individual components that contribute to the total cycles shows that the PWM and ADC are closely integrated and the trigger delay contributes to few additional cycles. The ADC early interrupt generation feature allows the ADC conversion to proceed in parallel while the system is responding to the interrupt. This allows the conversion cycles to overlap with the interrupt generation and compiler context save cycles, thus reducing the overall time required for responding to the ADC sampling event. In comparison, for any other system that does not have the ADC early interrupt generation feature, the interrupt response will include the conversion cycles thus increasing the overall response time. The other advantage of this feature is that ADC result can be applied to the system closer in time to when it was sampled resulting in more accurate system control.

3.2 Peripheral Access

The ACI real-time benchmark has two peripheral access responses: ADC read and convert to float as well as PWM write. For the response times for each of these operations, see Figure 3-3.



```
ACI Benchmark Test:
F28004x CPUCLK = 100000000 Hz, from RAM, using TMU


Execution in cycle counts (avg, max, min) over 1024 iterations

--------------------------------------------------------------------------
                                        AVG      MAX      MIN
 INT Response (trigger to ISR entry) :  49       53       47
 Read 2 ADC, convert float           :  3        3        3
 Clarke Transform                    :  11       11       11
 3 PID Controller Transforms         :  107      107      107
 Inverse Park Transform              :  23       23       23
 Flux Estimator                      :  167      168      166
 Speed Estimator                     :  66       67       66
 Park Transform                      :  18       18       18
 SVGen Transform                     :  75       82       66
 Write 3 PWM                         :  10       10       10
--------------------------------------------------------------------------
 Total                               :  529      542      517
--------------------------------------------------------------------------

Motor Speed at end of test (expected 0.5) : 0.500923

Test program end.
```

Figure 3-3. ACI Motor Benchmark Peripheral Access Benchmark Data

In this particular implementation, only two ADC inputs are being read and three PWM outputs are being written. As can be seen in Table 3-2, in other practical applications involving motor control or digital power the number of these operations can be larger and can have a bigger impact on the execution duration of a control application.

In the C2000 architecture the CPU is closely integrated with the peripherals and the instruction set architecture is optimized for floating point operations. From the benchmark data in Table 3-3, it can be derived that it takes only ~2 cycles to read one ADC input and convert it to float and just ~3 cycles to write one PWM output. As such, C2000 architecture can effectively scale to applications with large number of peripheral accesses.

Table 3-2 captures the estimated cycles for some real control applications based on the benchmarking data.

Table 3-2. Estimated Peripheral Access Cycles for Real Control Applications

Application	Number of ADC Reads	Number of PWM Writes	Estimated Total Cycles
3 Phase AC motor control application	7	3	23 [1]
Totem-pole PFC digital power application	10	9	47 [2]

[1] 7 ADC reads (2 cycles each) + 3 PWM writes (3 cycles each) = (7 x 2) + (3 x 3) = 23
[2] 10 ADC reads (2 cycles each) + 9 PWM writes (3 cycles each) = (10 X 2) + (9 X 3) = 47

3.3 TMU (math enhancement) Impact

Many C2000 targets have a math extension called Trigonometric Math Unit (TMU) ,which is an extension of the Floating-Point Unit (FPU) and enhances the instruction set of the C28x+FPU by efficiently executing trigonometric and arithmetic operations that are commonly used in control system applications.

The ACI motor control application has control algorithms: Park, Inverse Park and Flux estimator that involve trigonometric math. To demonstrate the impact that the TMU can make, the F28004x has two build configurations ***SignalChain_RAM_TMU*** and ***SignalChain_RAM_FastRTS***.

In the following figures, Figure 3-4 shows the execution time derived from the benchmark cycles data and Figure 3-5 shows application size (code and data) for the control code (driver libraries and runtime are excluded) as indicated in the compiler tools generated linker map file, for each of the two configurations.

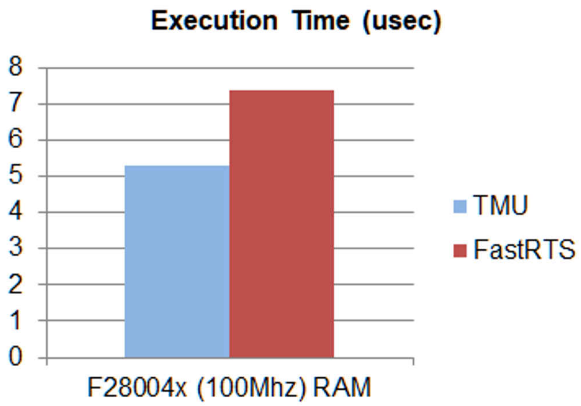


Figure 3-4. Control Loop Execution Time Out of RAM With TMU and FastRTS

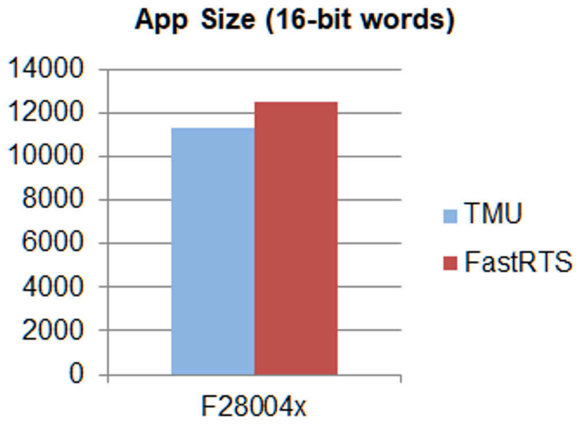


Figure 3-5. Application Size Compiled for TMU and FastRTS

The data in the graphs show that the TMU has a two-fold impact:

- **Faster Execution Time:** The execution of the application becomes faster by 28%.
- **Smaller Application Size:** A typical trigonometric implementation involves lookup tables and floating-point computations to calculate common trigonometric operations. The TMU provides instructions for these trigonometric operations and eliminates both the lookup tables and complex floating point code. This translates into smaller application size allowing smaller memory devices with TMU to be able to support the application. In this particular example the application size reduces by 14%.

3.4 Flash Performance

Many applications are configured to execute out of flash. As such flash performance is a critical criteria for measuring the capability of a real-time system. Typical benchmarks compute flash efficiency by executing synthetic code that may not reveal performance when executing a control application with its combinations of non-linear code flow, data tables and peripheral accesses.

The F28004x and F2837x have two build configurations: '***SignalChain_RAM_TMU***' and '***SignalChain_FLASH_TMU***'. The execution time for these configurations are displayed in Figure 3-6 and Figure 3-7.

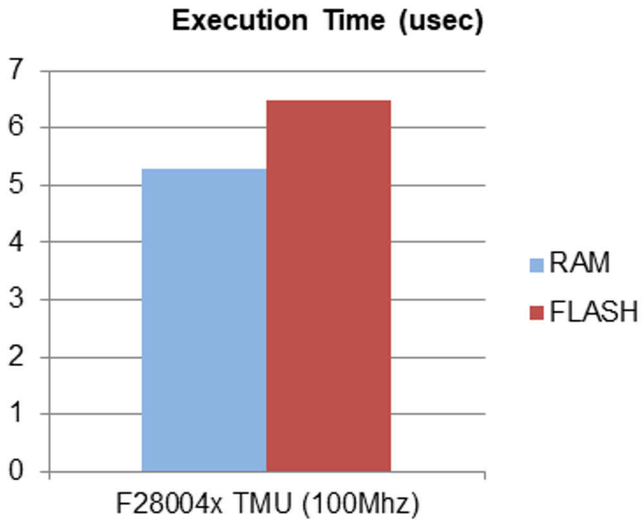


Figure 3-6. TMS320F28004x Execution Time From RAM and Flash

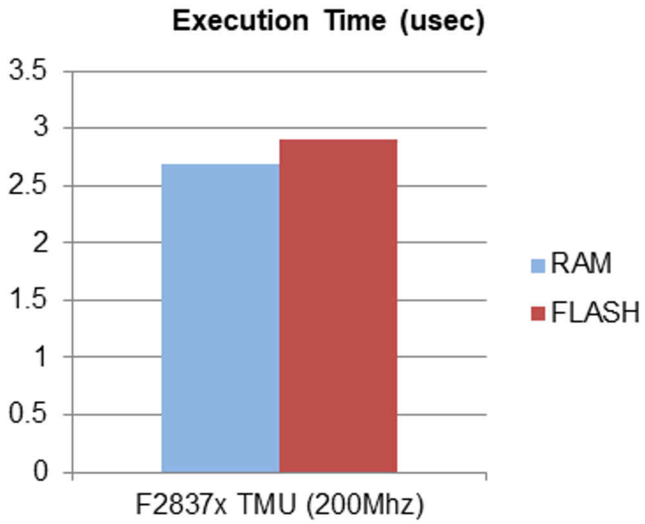


Figure 3-7. TMS320F2837x Execution Time From RAM and Flash

The flash performance as derived from the total average execution cycles benchmark data is listed in Table 3-3.

Table 3-3. Flash Execution Performance Compared to RAM

	RAM Execution (cycles)	Flash Execution (cycles)	Flash Relative Performance (RAM_cycles * 100 / Flash_cycles)	Flash Technology
TMS320F28004x	529	648	82%	• 100 MHz CPU Clock • 20 MHz Flash Speed • 4 Wait States
TMS320F2837x	537	582	92%	• 200 MHz CPU Clock • 50 MHz Flash Speed • 3 Wait States

As can be seen by the data in the table, the flash performance of C2000 devices for a real world application like ACI motor control is very good (~80-90%) when compared to the RAM performance.

3.5 Control Law Accelerator (CLA)

The CLA is a task driven fully-programmable independent 32-bit floating-point hardware accelerator that is designed for math intensive computations. The CLA also has access to control peripherals like ADC and PWM. The CLA supports eight hardware tasks. A hardware task can be initiated by the C28x CPU or by a control peripheral. More details on CLA can be found in the documentation linked in the References section. These capabilities make the CLA a powerful ally in achieving real-time system performance goals in two ways:

1. The CLA can be triggered by a peripheral and can execute the entire control loop thus freeing up the C28x CPU for other activities such as running communications stack and thus effectively doubling up the available compute power.
2. The C28x CPU can offload parts of the compute to the CLA that can reduce execution time and thus boost performance.

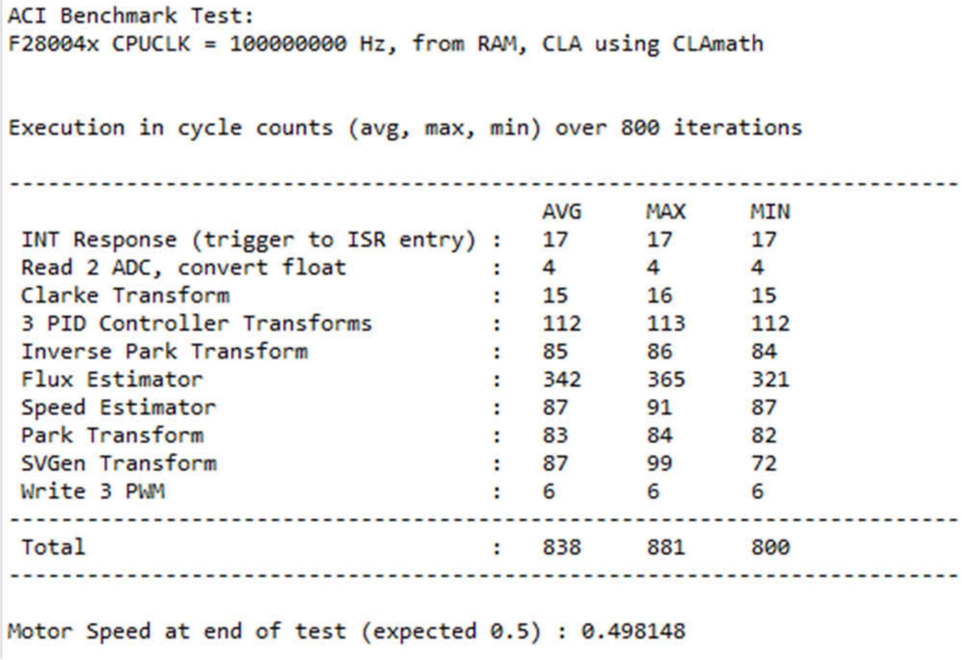
The ACI real-time benchmark has been ported to showcase both these usages of the CLA.

3.5.1 Full Signal Chain Execution on CLA

The CLA is a task driven architecture and has close integration with control peripherals. The device-specific TRM and data sheet have information on the peripherals that are accessible by the CLA.

The ADC can trigger a task on the CLA just like it triggers an interrupt on the C28x CPU. The CLA can also write to the PWM. These capabilities allow the entire ACI Signal Chain to be executed on the CLA with the ADC triggering the CLA task and the CLA task executing the entire control loop. While the CLA is executing the signal chain, the C28x CPU is idle. This demonstrates that in a more comprehensive example, the application can be partitioned to run the control loops from CLA and have the C28x CPU run other operations like communications stacks.

For the implementation, see the '*SignalChain_RAM_CLAmath_CLA*' build configuration.



```
ACI Benchmark Test:
F28004x CPUCLK = 100000000 Hz, from RAM, CLA using CLAmath


Execution in cycle counts (avg, max, min) over 800 iterations

-----------------------------------------------------------------------
                                         AVG      MAX      MIN
 INT Response (trigger to ISR entry) :   17       17       17
 Read 2 ADC, convert float           :   4        4        4
 Clarke Transform                    :   15       16       15
 3 PID Controller Transforms         :   112      113      112
 Inverse Park Transform              :   85       86       84
 Flux Estimator                      :   342      365      321
 Speed Estimator                     :   87       91       87
 Park Transform                      :   83       84       82
 SVGen Transform                     :   87       99       72
 Write 3 PWM                         :   6        6        6
-----------------------------------------------------------------------
 Total                               :   838      881      800
-----------------------------------------------------------------------

Motor Speed at end of test (expected 0.5) : 0.498148
```

Figure 3-8. ACI Motor Benchmark Output for CLA Executing Full Signal Chain

The insights from the benchmarking are detailed as follows.

3.5.1.1 CLA ADC Interrupt Response Latency

The CLA is a task driven architecture and hence does not have context save and restore overheads of a traditional interrupt architecture. As a result the CLA has a low interrupt latency that allows it to read the ADC samples "just in time". More information on this topic can be found in the *CLA* section of the device-specific TRM.

In the application, the ADC setup to trigger the CLA Task is the same as the configuration where the ADC triggers the C28x interrupt. Table 3-4 shows how the CLA compares to the C28x and demonstrates the low latency in the absence of any context save overhead.

Table 3-4. ADC Interrupt Response Latency for CLA and C28x on F28004x

Compute Core	ADC Interrupt Response (cycles)
CLA	17
C28x	47

3.5.1.2 CLA Peripheral Access

The CLA is closely integrated with the ADC and PWM peripherals just like the C28x CPU. The peripheral access latency for CLA is similar to C28x CPU and as such the CLA is also similarly efficient. This efficiency is critical for control loop execution.

Table 3-5. Peripheral Access Latency for CLA and C28x on F28004x

Compute Core	2 ADC reads (cycles)	3 PWM writes (cycles)
CLA	4	6
C28x	4	9

3.5.1.3 CLA Trigonometric Math Compute

The CLA is efficient at floating point compute. However, unlike the C28x CPU which has the TMU hardware accelerator, the CLA does not have a trigonometric hardware accelerator and requires a software implementation. The C2000Ware software package provides CLAmath library that has efficient implementation of trigonometric operations. In the ACI application, the CLAmath library APIs are used for Park, Inverse Park and Flux Estimator compute blocks that contain trigonometric operations.

Table 3-6. CLAmath Performance for F28004x

Compute Core	Park/Inverse Park (cycles)	Flux Estimator (cycles)
CLA	83/85	342
C28x	18/24	167

It is easy to port code created for C28x CPU utilizing TMU operations to CLA. The C28x TMU compiler intrinsics map to CLAmath libray functions when compiled for CLA. As such the ACI benchmark control code running on C28x CPU with TMU did not need to be modified to run from CLA and only required linking the CLAmath library and including the CLAmath.h header file.

3.5.2 Offloading Compute to CLA

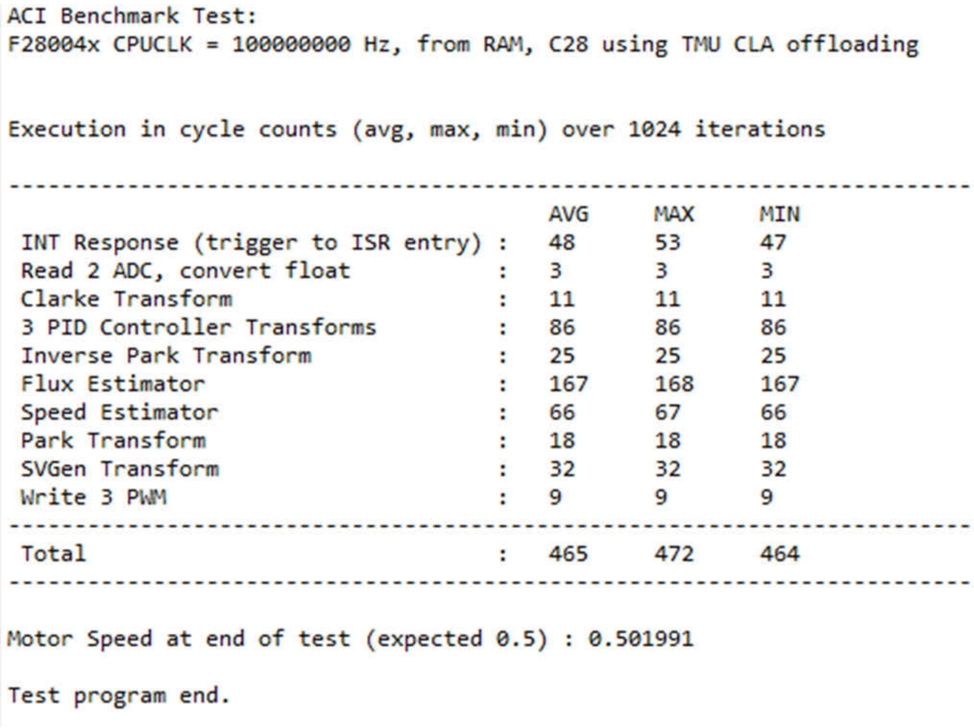
The C28x CPU can software trigger a CLA task using the IACK instruction. This can be used by the C28x to offload computations to the CLA at specific points in the code running on the C28x CPU. The ACI benchmark example is fairly linear in that one control algorithm block requires an input from a previous control algorithm block. However there are two instances where parallelism can be introduced:

1. PID control block:

   There are three instances of the PID control algorithm and one of these (PID Id instance) does not have dependencies on the others and can be parallelized. In the implementation, the C28x offloads the PID Id control execution to the CLA Task 2 while the C28x is executing the PID for Speed and Iq in parallel.
2. SVGen control block:

   The SVGen control block depends on the output of the Inverse Park transform. Since this is a sensor-less ACI implementation additional control blocks are also present such as Flux and Speed Estimators. The SVGen is not dependent on these and can be parallelized with the Estimators. In the implemetation the C28x offloads the SVGen calculation to the CLA Task 1.

For the implementation, see the '*SignalChain_RAM_TMU_CLA_OFFLOAD*' build configuration. The implementation demonstrates how easy it is to transition code to offload compute to the CLA. The same PID inline C function header file is included on the C28x source file as well as CLA source file and gets compiled into C28x code on C28x side and CLA code on CLA side.



```
ACI Benchmark Test:
F28004x CPUCLK = 100000000 Hz, from RAM, C28 using TMU CLA offloading


Execution in cycle counts (avg, max, min) over 1024 iterations

-------------------------------------------------------------------------
                                       AVG      MAX      MIN
 INT Response (trigger to ISR entry) : 48       53       47
 Read 2 ADC, convert float           : 3        3        3
 Clarke Transform                    : 11       11       11
 3 PID Controller Transforms         : 86       86       86
 Inverse Park Transform              : 25       25       25
 Flux Estimator                      : 167      168      167
 Speed Estimator                     : 66       67       66
 Park Transform                      : 18       18       18
 SVGen Transform                     : 32       32       32
 Write 3 PWM                         : 9        9        9
-------------------------------------------------------------------------
 Total                               : 465      472      464
-------------------------------------------------------------------------

Motor Speed at end of test (expected 0.5) : 0.501991

Test program end.
```

Figure 3-9. ACI Motor Benchmark Output for C28x With CLA Offloading

The opportunity to offload compute to CLA resulted in an execution cycle count reduction thus boosting performance by 12%.

4 C2000 Value Proposition

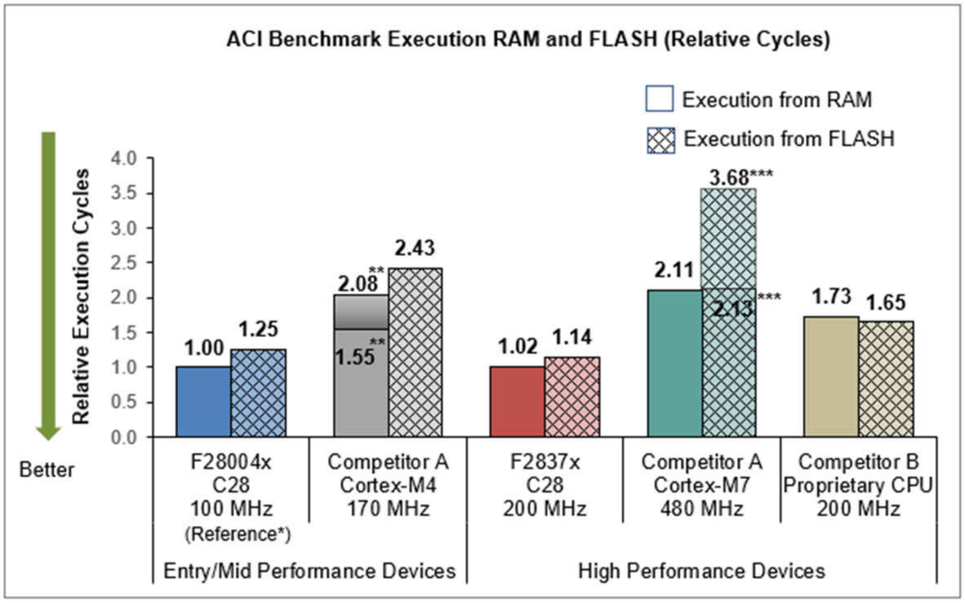
The ACI motor control benchmarking application is self-contained and does not require any special external hardware to execute. Hence, this application can be easily ported to any device to evaluate real-time control performance. A multi-device analysis demonstrates that the C2000 platform offers a rich feature set with an optimized architecture which leads to excellent performance across the real-time signal chain.

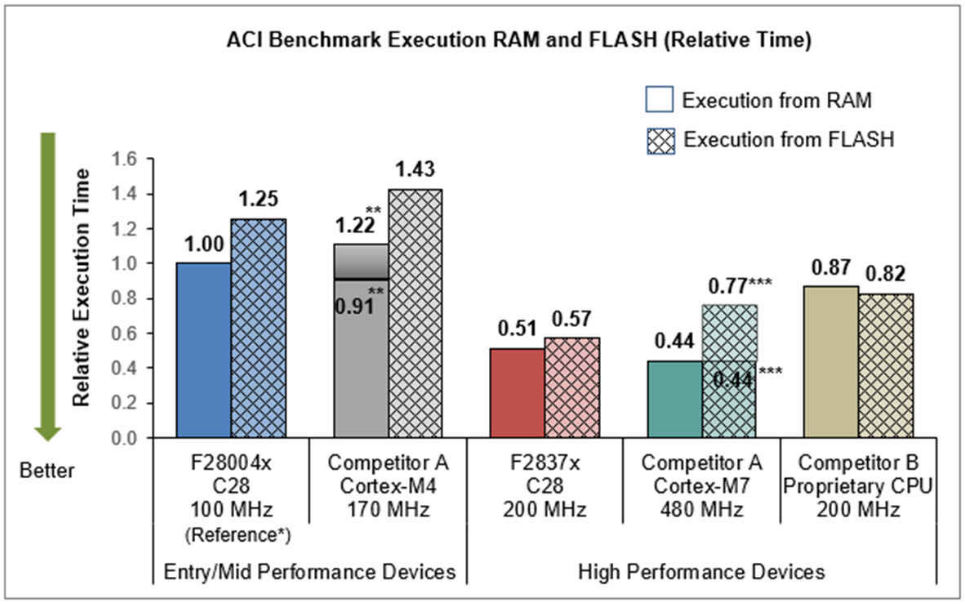
Note

1. The multi-device analysis for TI C2000 and Competition A and B devices presented in this chapter is based on actual measured data compiled by TI internally in 2021 with ACI motor benchmark example in C2000Ware 3.04.00.00 ported to these devices.
2. All of the measurements were performed by setting the tool chain to compile code for maximum performance/speed and operating the devices at the maximum CPU frequency.
3. The cycles/time was measured for each of the devices and the numbers were compared against the F28004x executing from RAM to determine relative performance. The relative performance is represented in each of the graphs and this is why F28004x is marked as "Reference" and is always 1.0.
4. In the graphs, a lower relative number indicates fewer cycles or lesser time and hence better performance.
5. The classification of the categories in the graph as "Entry/Mid performance" and "High performance" is based on TI nomenclature. The competition devices are placed in the same category as the TI device that competes with the competition device.

4.1 Efficient Signal Chain Execution With Better Real-Time Response Than Higher Computational MIPS Devices

The ACI benchmark data reveals that C2000 devices are able to quickly respond to a sampling event and efficiently execute a response thus resulting in a ***fast sample-to-output response***. This also allows for more available MIPS for additional operations. With a ***good flash execution efficiency*** (RAM execution time/Flash execution time) of ***82% for F28004x*** (entry/mid performance device) and ***92% for F2837x*** (high performance device), execution out of flash is also a viable option to meet real-time performance requirements.





* As indicated in the Note, F28004x execution from RAM is the reference.

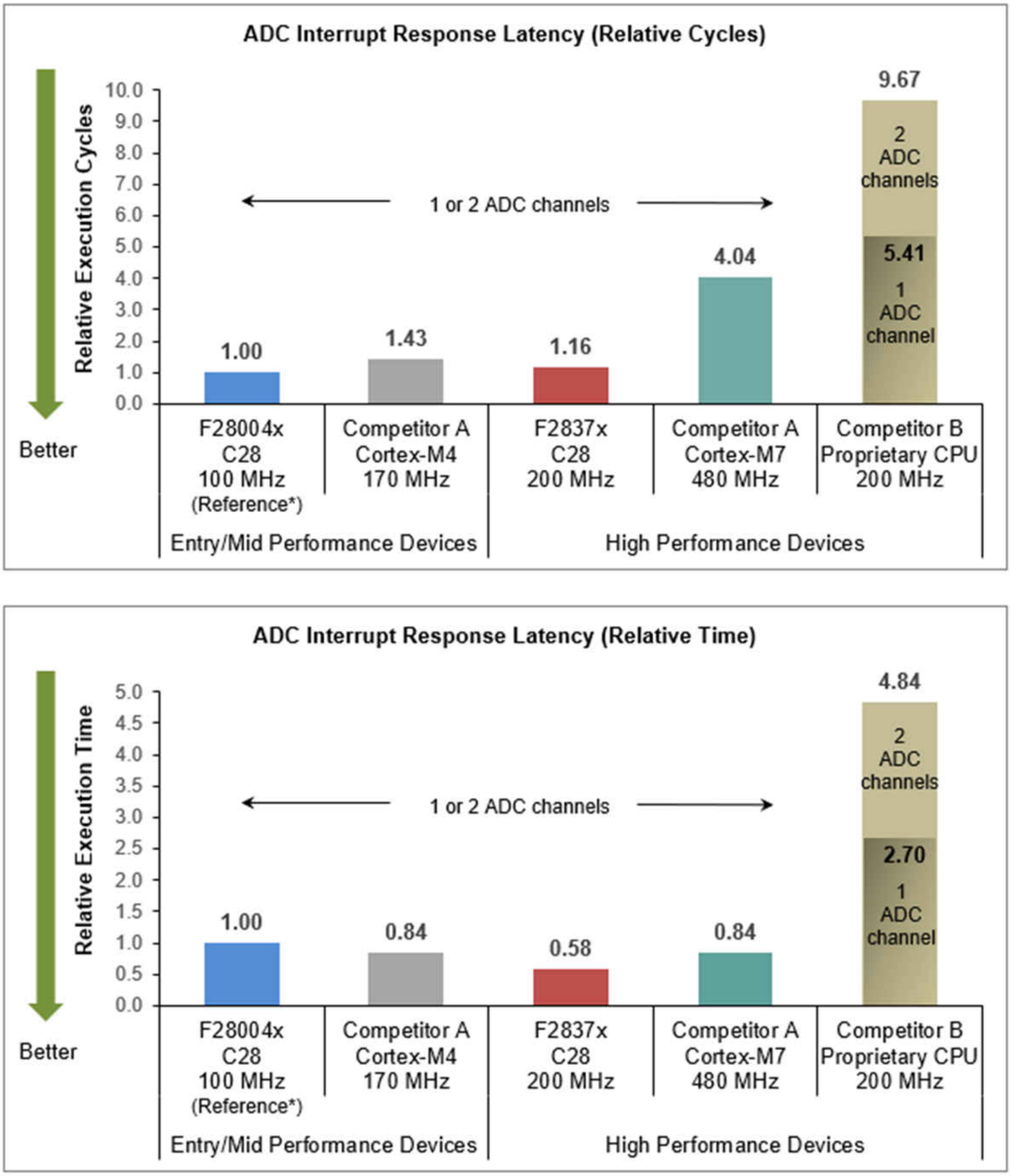
** Best case and worse case performance numbers using faster RAM or standard RAM banks.

*** Best case and worse case performance with warm and cold cache

Figure 4-1. ACI Motor Control Benchmark Execution (relative cycles and relative time)

4.2 Excellent Real-Time Interrupt Response With Low Latency

The C2000 devices have excellent analog peripherals. The ***ADCs have a fast sampling rate*** and ***early interrupt generation feature*** where in an interrupt can be triggered while the first ADC channel value in a sequence is still being converted. Together with an ***efficient interrupt architecture*** this translates to fewer cycles when responding to a trigger to an ADC sampling event.

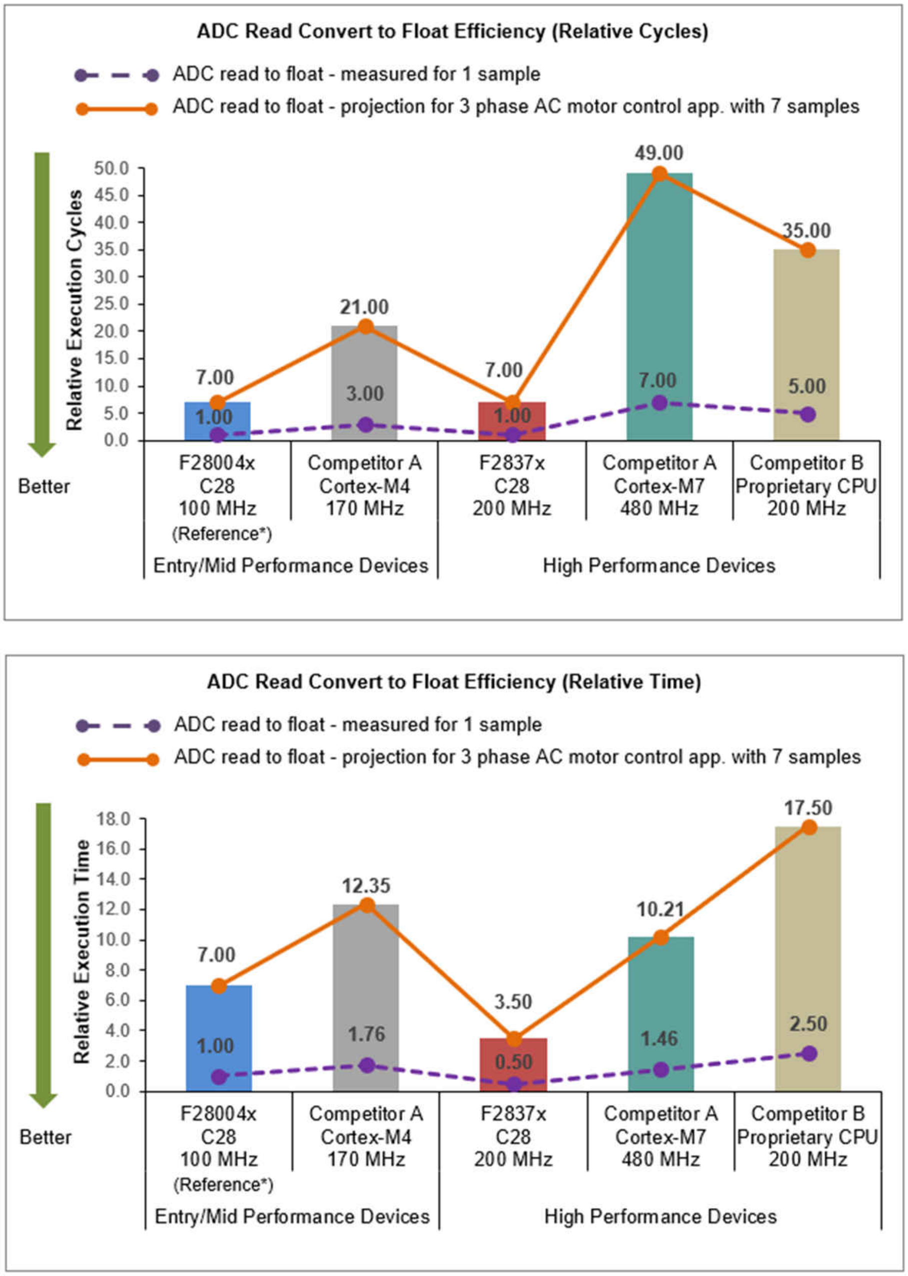


* As indicated in the Note, F28004x execution from RAM is the reference.

Figure 4-2. ADC Interrupt Response Latency (relative cycles and relative time)

4.3 Tight Peripheral Integration That Scales Applications With Large Number of Peripheral Accesses

The C2000 instruction set and tight peripheral integration make it possible to read an ***ADC result and convert to float*** in just ***2 cycles***. As such the overhead for ADC access is extremely small and hence highly scalable to applications with a large number of ADC accesses.

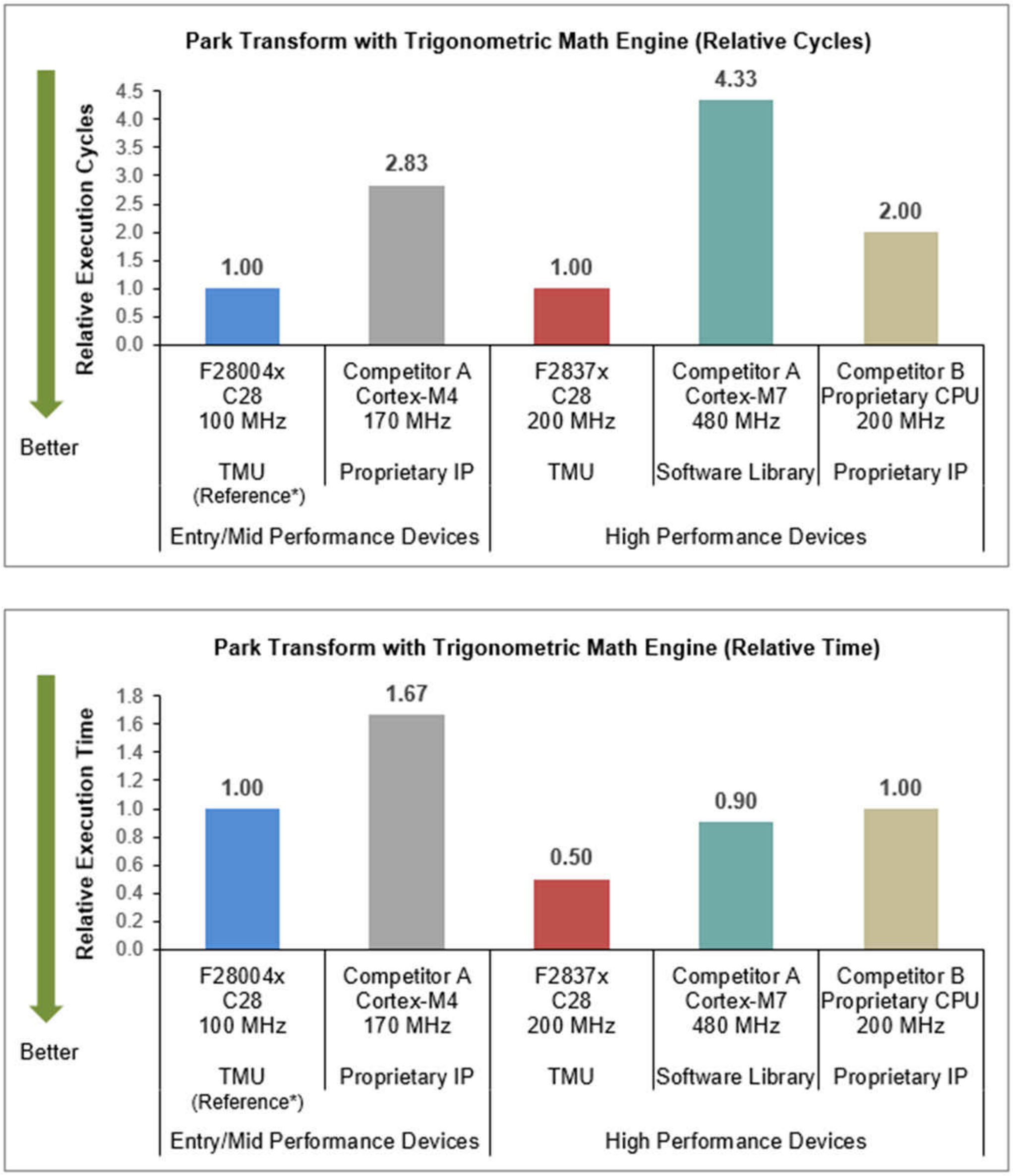


* As indicated in the Note, F28004x execution from RAM is the reference.

Figure 4-3. ADC Read and Convert to Float Efficiency (relative cycles and relative time)

4.4 Best in Class Trigonometric Math Engine

Trigonometric operations are common in many control algorithms, Park transform is one such example. Efficient execution of trigonometric operations can be vital for minimizing compute duration. All Gen-3 ***C2000 devices across a wide performance range*** from entry/mid performance devices like F28004x to high performance devices like F2837x have a trigonometric math engine called ***Trigonometric Math Unit (TMU).*** The TMU extends the instruction set for trigonometric operations. With an ***easy to use*** programming model through the support of compiler intrinsics, it is easy to apply the performance boost of the TMU to real-time control algorithms executing on C2000 devices.



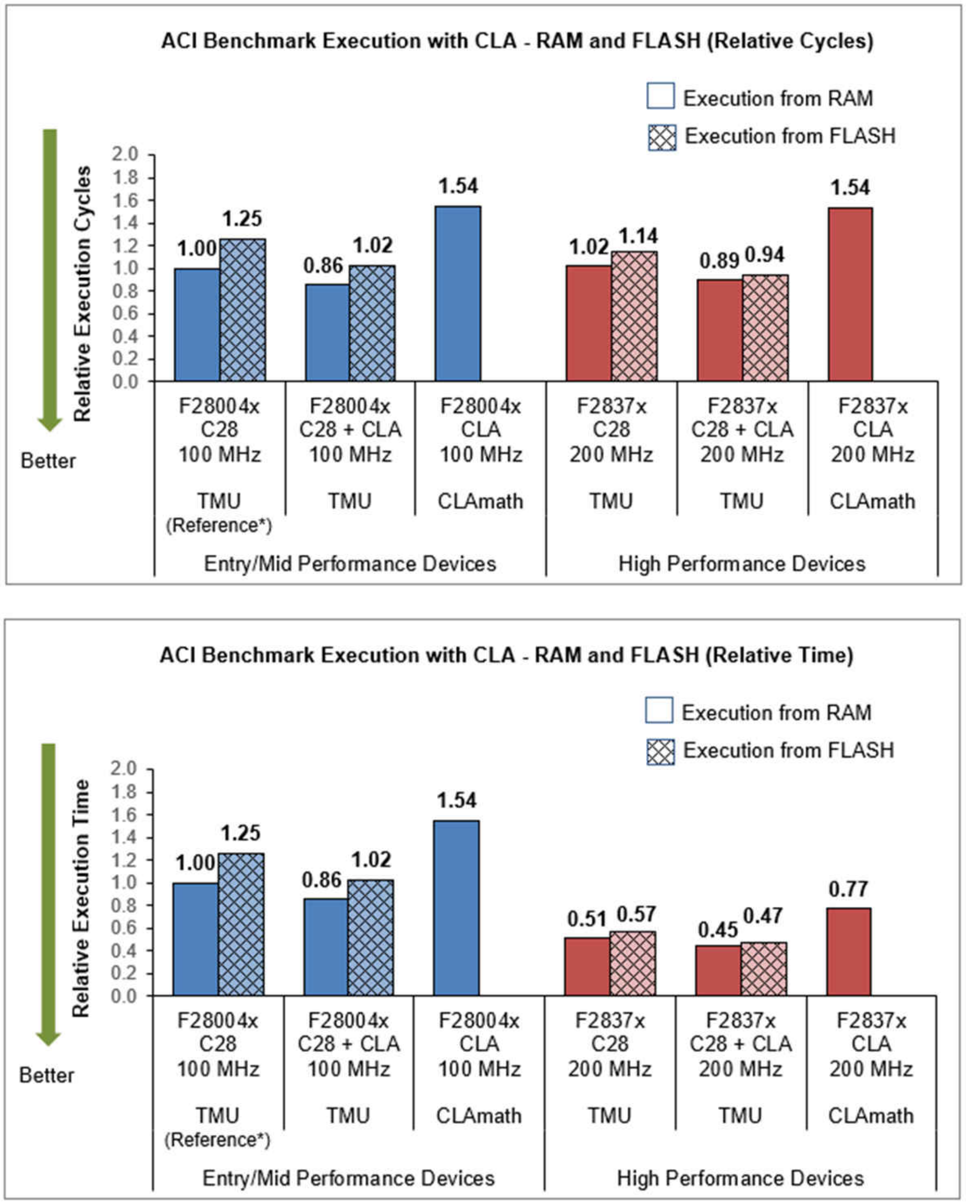
* As indicated in the Note, F28004x execution from RAM is the reference.

Figure 4-4. Park Transform With Trigonometric Math Engine (relative cycles and relative time)

4.5 Versatile Performance Boosting Compute Engine (CLA)

Many C2000 devices have a CLA (Control Law Accelerator) which runs at the same frequency as the C28x CPU and can be utilized in two different ways to achieve real-time application goals.

- CLA executing full signal chain - CLA executing independent of C28x CPU, freeing up C28x CPU for other activities and effectively doubling the available MIPS.
- C28x CPU offloading compute to CLA - parallelism resulting in better execution performance by reducing the sample to output response time.



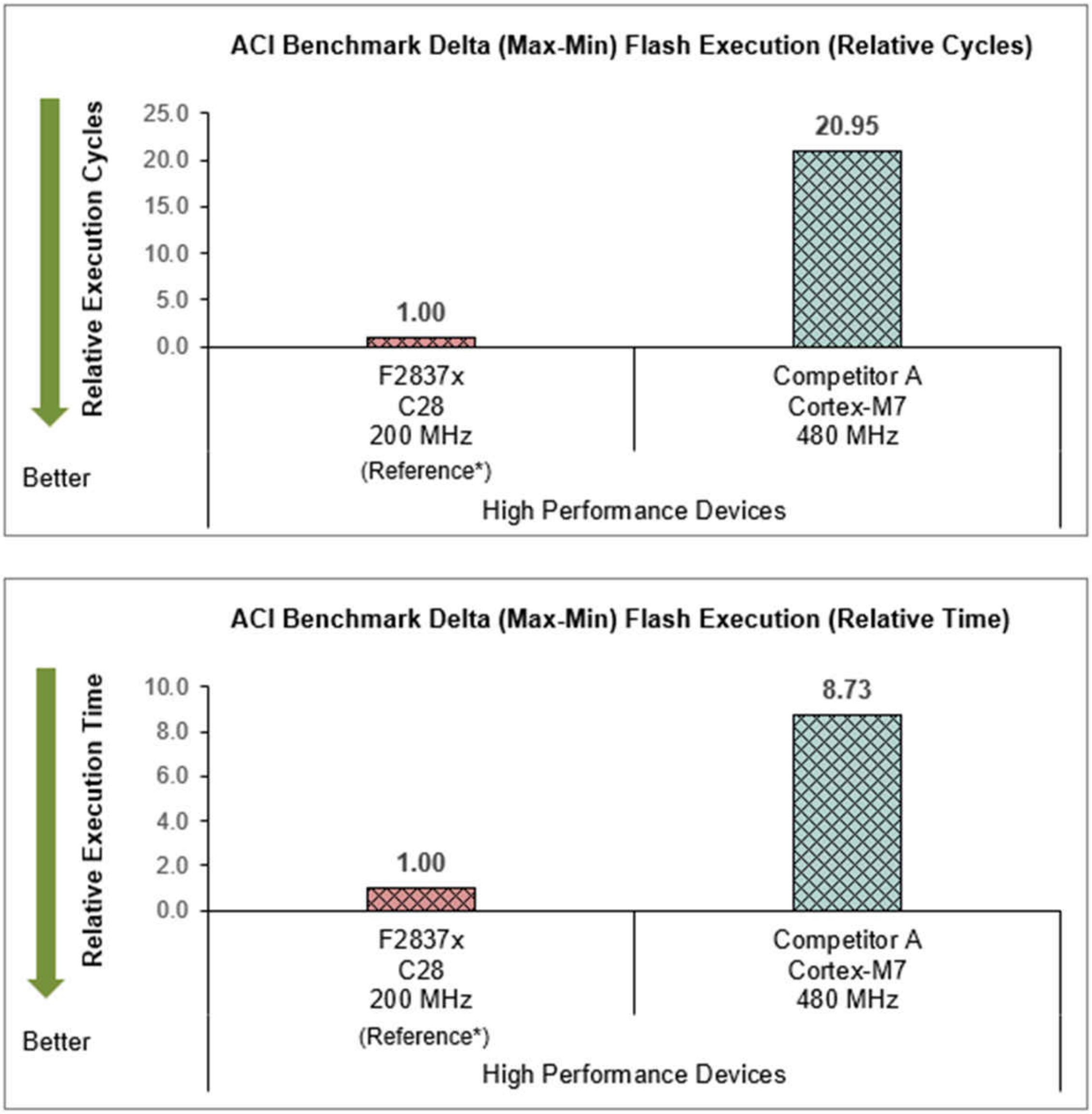
* As indicated in the Note, F28004x execution from RAM is the reference.

CLA runs from RAM only and hence does not have a Flash execution data point.

Figure 4-5. ACI Motor Control Benchmark CLA Execution (relative cycles and relative time)

4.6 Deterministic Execution due to Low Execution Variance

Deterministic execution is highly desired for real-time systems. The C2000 family of devices have tiny prefetch buffers for improving Flash execution but do not have any caches. While caches can improve performance, they can also introduce a large variance in execution duration due to cache operations resulting from cache misses and coherency operations. The C2000 devices perform well without caches and in turn also allow deterministic execution with a low execution variance.



* F2837x execution from Flash is the reference.

Figure 4-6. ACI Motor Control Benchmark Execution Variance from Flash (relative cycles and relative time)

5 Summary

This application note demonstrates that typical industry software benchmarks that focus on processing time alone do not effectively convey the true performance of a system when executing real-time control applications. Real-time benchmarking approach gives a more comprehensive and realistic picture of the real-time capabilities of a system. For example, the ACI motor benchmark data for F28004x device with TMU execution from RAM shows that the non-control-algorithm components (INT response + Read ADC + Write PWM) contribute about 12% of the total signal chain execution duration which would be unaccounted for in a typical software benchmark. This percentage will vary for different devices depending on the architecture and analog capabilities and can be higher for a device where the peripheral accesses or ADC sampling takes longer as well as for other real-time control applications with a greater number of peripheral accesses.

The real-time benchmarking data in this application note showcases the differentiated features of the C2000 architecture such as tight peripheral integration, ADC early interrupt feature, optimized instruction set with math enhancement like TMU and efficient flash technology that together help reduce the sample to output response time and thus make the C2000 MCUs highly optimized for real-time control applications.

6 References

- Texas Instruments: *TMS320F28004x Microcontrollers Technical Reference Manual*
- Texas Instruments: *TMS320F28004x Microcontrollers Data Sheet*
- Texas Instruments: *TMS320F2837xD Dual-Core Microcontrollers Technical Reference Manual*
- Texas Instruments: *TMS320F2837xD Dual-Core Microcontrollers Data Sheet*
- Texas Instruments: Software Development Kit - C2000Ware for C2000 MCUs
- Texas Instruments: TMU instruction set - *TMS320C28x Extended Instruction Sets Technical Reference Manual*
- Texas Instruments: TMU additional details - *Enhancing the Computational Performance of the C2000™ Microcontroller Family*
- Texas Instruments: CLA details - *Enhancing the Computational Performance of the C2000™ Microcontroller Family*
- Texas Instruments: The Essential Guide for Developing With C2000™ Real-Time Microcontrollers
- Texas Instruments: CLA Application Report - Software Examples to Showcase Unique Capabilities of TI's C2000™ CLA

7 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Revision * (April 2021) to Revision A (December 2021)	Page
• Update was made in the Abstract of this document.	1
• Updated the numbering format for tables, figures and cross-references throughout the document.	3
• Update was made in Section 1	3
• Update was made in Section 2.1	4
• Update was made in Section 2.2	4
• Update was made in Section 2.3	5
• Update was made in Section 2.5.1	8
• Update was made in Section 2.6	8
• Update was made in Section 3	9
• Update was made in Section 3.1	10
• Update as made in Section 3.2	11
• Added new Section 3.5	14
• Update was made in Section 4.1	18
• Update was made in Section 4.2	19
• Update was made in Section 4.3	20
• Update was made in Section 4.4	21
• Added new Section 4.5	22
• Added new Section 4.6	23
• Update was made in Section 5	24
• Update as made in Section 6	24

IMPORTANT NOTICE AND DISCLAIMER

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265