TriCore Family

AP32168

Application Performance Optimization for TriCore V1.6 Architecture

Application Note

V1.0 2011-03

Microcontrollers

Edition 2011-03

Published by
Infineon Technologies AG
81726 Munich, Germany

Device1
Revision History: V1.0, 2011-03

Previous Version: none

Page	Subjects (major changes since last revision)

Table of Contents

1 Introduction

TriCore, the new processors family generation based on TC1.6 core provides high performance architecture for embedded applications. To fully utilize its capability, the necessary steps involving hardware configuration and software optimization are described for achieving the best application performance.

Hardware configuration is a well defined task of parameters setting appropriate for defined target system. Completeness and correctness of these settings are crucial for best performance. Software optimization is iterative process of compiler/linker settings with no clear convergence to optimum. Interaction between various settings and not unique best selection satisfying all application functions determine the optimization complexity.

This application note will guide you through the optimization process to achieve best application performance. Included application benchmarks results will demonstrate the effect of various optimization settings.

1.1 Naming Convention

TC1.6	-	TriCore Architecture V1.6
TC1.3.1	-	TriCore Architecture V1.3.1
TC1793	-	Processor based on TC1.6 Architecture
TC1797	-	Processor based on TC1.3.1 Architecture
SFR-R	-	TriCore registers stored in Altium TASKING install directory under regtc179x.sfr

1.2 Tool chain and Target Hardware

- All included tool chain configuration data are based on Altium TASKING VX-toolset for TriCore V3.4r1.
- Target Hardware for performance evaluation - TC1793 and TC1797 TriBoards

1.3 Application Benchmarks

Included

- APP-1 Application with code size smaller than cache size, linear code with limited loops.
- APP-2 Application with code size smaller than cache size, including many loops.
- APP-3 Complete application with code size significantly exceeding the cache size. Intensive use of load store operation and integer arithmetic.
- APP-4 DSP fixed point algorithm.

2 Application Performance Optimization

2.1 Performance Criteria

In this document performance optimization and measurement will focus on

- Code size
- CPU execution time

Performance evaluation is used to:

- Demonstrate that the system meets performance criteria.
- Compare two systems to find which performs better.
- Measure what parts of the system or workload cause the system to perform badly

In performance testing, it is often crucial for the test conditions to be similar to the expected actual use. Included performance data is primary based on real applications expected to be implemented on described devices.

2.2 Measuring Performance using TriCore Performance Counters

Real-time measurement of core performance provides useful insights to system developers, compiler developers, application developers and OS developers. TriCore includes the ability to measure different performance aspects of the processor without any real-time effect on its execution.

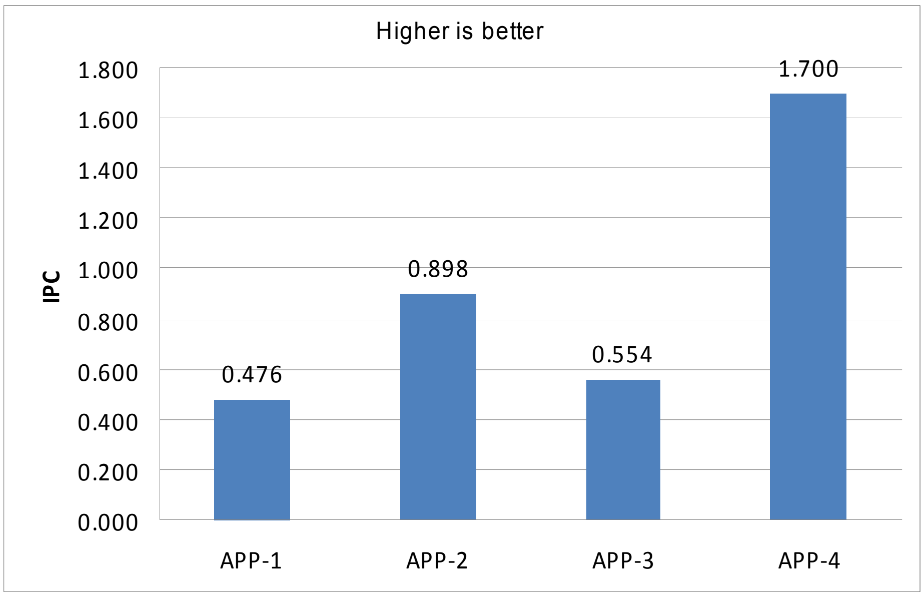
Following performance counters are integrated in the TriCore core module:

- Dedicated counters
  - CCNT: CPU Clocks Counter
  - ICNT: Instruction Counter
- Configurable counters, each with selectable one of four count modes
  - M1CNT:
    1. IP_DISPACTH_STALL, 2. PCACHE_HIT, 3.DCACHE_HIT, 4.TOTAL_BRANCH
  - M2CNT:
    1. LS_DISPACTH_STALL, 2. PCACHE_MISS, 3. DCACHE_MISS_CLEAN, 4. PMEM_STALL
  - M3CNT:
    1. LP_DISPACTH_STALL, 2. MULTI_ISSUE, 3. DCACHE_MISS_DIRTY, 4. DMEM_STALL

You can use directly all the available performance counters to measure execution time, instruction count or other values. Alternatively dedicated performance analyzing tools integrated within debuggers can be used.

Instruction Per Cycle (IPC) defined as ratio of (Number of Instruction)/(CPU Clocks) is very useful as a measure of optimization progress. It reflects a compiler configuration quality, efficiency of memory system and code/data location.

Figure 1 shows measured IPC values of some representative benchmarks using TriCore Performance Counters and calculated by ratio of ICNT/CCNT. This measurement is very accurate and can be evaluated without any special equipment.



```
// Start Measure
   t1a=__mfcr(CCNT);
   t1b=__mfcr(ICNT);

   Benchmark();

// Stop Measure
  t2a=__mfcr(CCNT);
  t2b=__mfcr(ICNT);
  cpu_clk = t2a-t1a;
  instr_cnt = t2b-t1b;


// CCNT, ICNT are
// defined in SFR file
```

Figure 1 IPC Measurement with TriCore Performance Counters

*Note: Performance Counters need be enabled before can be used.
To enable all the counters set the CCTRL.CE bit to 1.*

3 Hardware Configuration

Hardware configuration is a well defined task of parameters setting, appropriate for defined target system, including:

- CPU and other clocks frequencies
- Flashes wait states
- ICACHE and DCACHE configuration

3.1 CPU Clock

CPU Clock is the most fundamental configuration parameter having impact on the other system settings and a primary factor determining execution time.

Figure 2 shows the clock control unit providing separate clocks for several TriCore modules. Each clock frequency is configurable by the dedicated divider driven by PLL or PLL_ERAY clocks. Output frequency is the result of the input frequency and the divider value. Due to common input clock the output frequency to each module is not free selectable. Maximal allowable frequencies and some defined constraint determine the possible clocks frequencies.

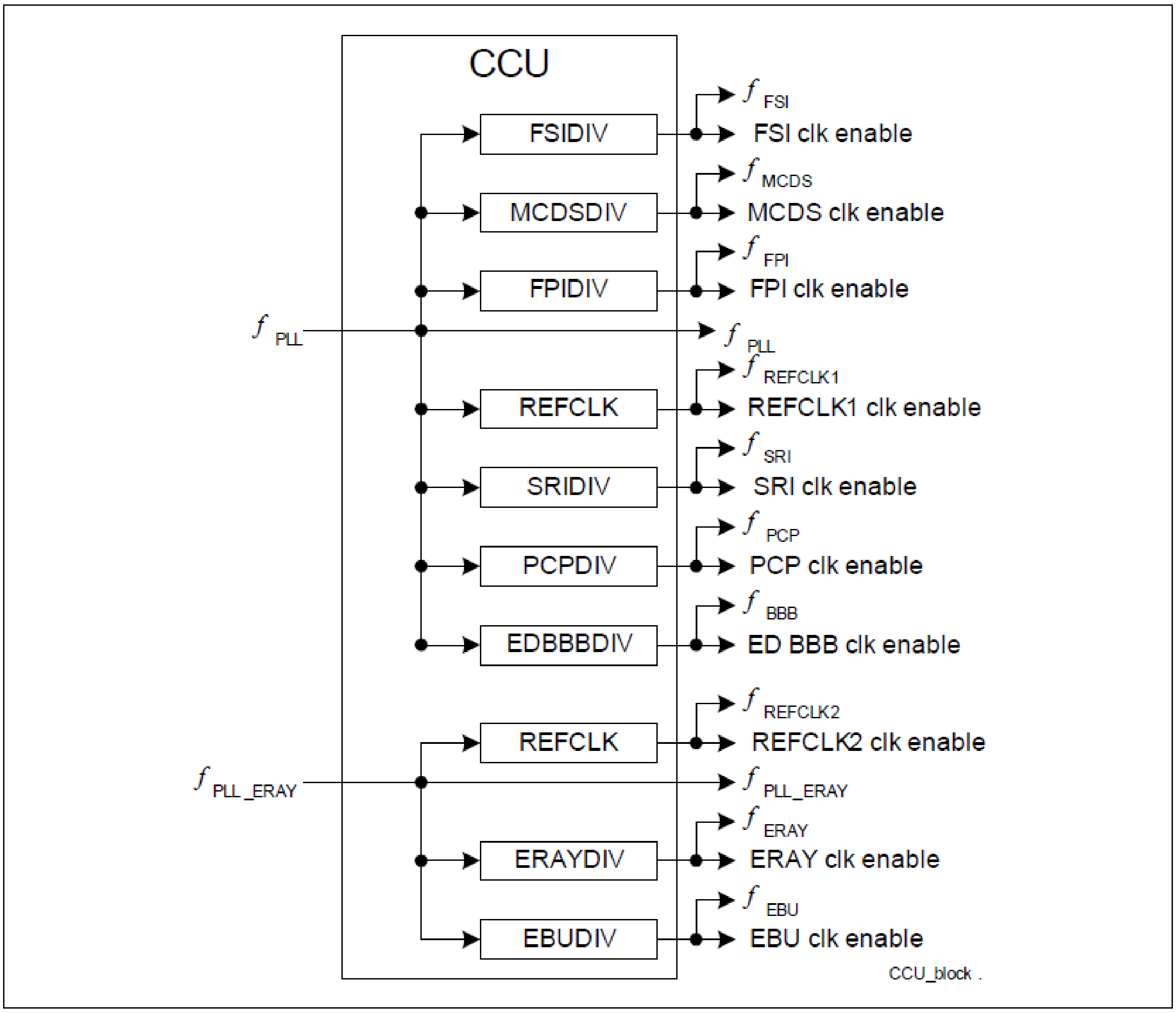


Figure 2 Clock Control Unit

Rules for calculation of modules clocks:

• fCPU: fFSI	1:1 or 2:1
• fCPU: fFPI	n:1 (n = 1...16)
• fPCP: fFPI	1:1 or 2:1
• fFPI: fMCDS	1:n or special case fFPI : fMCDS = 2:3
• fCPU: fSRI	1:1
• fMCDS: fCPU	1:2; 1:1 or 2:1
• fBBB: fMCDS	1:1 or 1:2

Table 1 Examples of modules clocks configuration

fPLL [MHz]	SRIDIV	fCPU [MHz]	FSIDIV	fFSI [MHz]	PCPDIV	fPCP [MHz]	FPIDIV	fFPI [MHz]
600	1	300	3	150	2	200	5	100
520	1	260	3	130	2	173.3	5	86.6
200	0	200	1	100	0	200	1	100

Note: In the ***Table 1*** *to have the real clock divider values you need add 1 to the table FSDIV values (e.g. FSIDIV = 3 means divide by (3+1))*

Note: Configuration of PLL Module used for generation of CPU Clock involves correct sequence of register configuration, monitoring of lock condition and final configuration.

3.2 Memory System

The gap between processor and memory performance is steadily growing. To compensate this speed differences, primary between PMU Flash units and the TriCore CPU, two 16KB caches ICACHE for instruction and DCACHE for data are part of TriCore memory hierarchy. 32KB Program Scratchpad RAM (PSPR) and 128KB Data Scratchpad RAM (DSPR) provides a fast, deterministic program fetch and data access for use by performance critical code sequences.

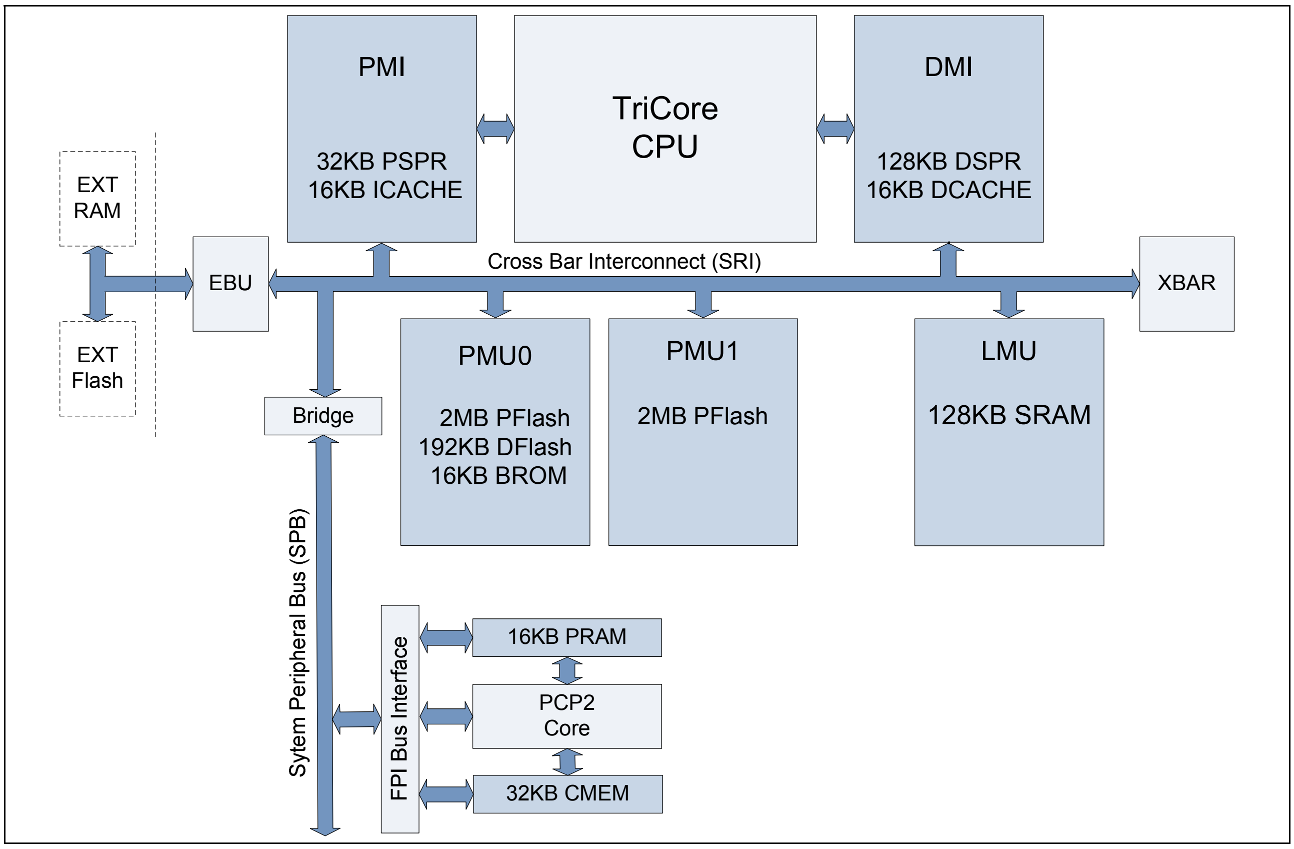


Figure 3 TriCore Memory Hierarchy (TC1.6)

Table 2 contains additional details of available TriCore memories, organized from the fasted to slowest. “Memory Config. WS” column includes a memory configurable wait states which should be set according to working conditions. Segment NC/C (not cacheable/cacheable) column contains segments addressable by each memory and whether it’s cacheable or not.

Table 2 TriCore TC1.6 Memories (Start from fastest)

Memory Type	Memory Config. Wait States (WS)	Segment NC* Segment C	Remarks
PSPR: Program Scratch-Pad RAM	-	0xC -	Used for deterministic and performance critical code (e.g. DSP Algorithms, OS)
ICACHE: Instruction Cache	-	-	Reduce significantly average access time for memories using cacheable segments
DSPR: Data Scratch-Pad RAM	-	0xD -	Used for deterministic and performance critical data access (e.g. DSP alg., OS)
DCACHE: Data Cache	-	-	Reduce significantly average access time for memories using cacheable segments
PMU0 - PFlash: Program Flash	1..15	0xA 0x8	See WS Table 3
PMU1 - PFlash: Program Flash	1..15	0xA 0x8	See WS Table 3
LMU SRAM	-	0xB 0x9	Used as overlay RAM, general data and code
EXT RAM	Memory type dependent	0xA 0x8	To be set according to used memory type and EBU frequency
EXT FLASH	Memory type dependent	0xA 0x8	To be set according to used memory type and EBU frequency

*NC –Not cacheable, C- Cacheable

3.2.1 PMU0 and PMU1 Flash memories

CPU and PMU-Flashes are connected to separate clocks which are derived from the same PLL output but using dedicated clock dividers SRIDIV and FSIDIV. Flashes and CPU can use the same clock frequency if it doesn't exceed 150 MHz, otherwise higher FSIDIV value need be used because FSI frequency is limited to 150MHz.

The required number of wait states for an initial access to PFlash or DFlash is related to the maximum FSI frequency. Because the default after reset is a worst case setting sufficient for all frequencies, the access times have to be configured by the user according to the application's frequency for optimum performance. This configuration of wait states (in number of FSI clock cycles) must be configured via the 4-bit-fields "WSPFLASH" and "WSDFLASH" in register FCON (Flash Configuration Register).

Table 3 includes example configurations for some selected CPU frequencies of the TC1.6 derivatives. PLL frequency determines the FSIDIV value which is selected to keep the FSI frequency below 150MHz.

Table 3 PMU0, PMU1 Flash wait states (WS) with PFLASH Ta=26 ns and DFLASH Ta=50 ns

fPLL [MHz]	SRIDIV	fCPU [MHz]	FSIDIV	fFSI [MHz]	PFlash WS Computed	PFlash WS Rounded	DFlash WS Computed	DFlash WS Rounded
600	1	300	3	150	3.9	**4**	7.5	**8**
520	1	260	3	130	3.4	**4**	6.5	**7**
200	0	200	1	100	2.6	**3**	5.0	**5**

*Note: The calculated wait states are relative to the fFSI. To calculate the wait states relative to fCPU WScpu =(fCPU / fFSI) * WS. e.g. PFlash WS relative to CPU are 8, 8, 6*

3.2.1.1 PMU0 and PMU1 Flashes wait states configuration

To configure **FSI** clock divider use **CCUCON0** register (SFR-R **SCU_CCUCON0)**

- FSIDIV = [19:16] bits (Divide Value = FSIDIV+1, e.g. FSIDIV=3 means divide by 4)

To configure **PMU0** Program and Data Flash wait states use **FLASH0** register **FCON** (SFR-R **FLASH0_FCON)** bit fields:

- WSPFLASH = [3:0] bits
- WSECPF = [4] bit
- WSDFLASH = [11:8] bits
- WSECDF = [12] bit

To configure **PMU1** Program and Data Flash wait states use **FLASH1** register **FCON** (SFR-R **FLASH1_FCON)** bit fields:

- WSPFLASH = [3:0] bits
- WSECPF = [4] bit
- WSDFLASH = [11:8] bits
- WSECDF = [12] bit

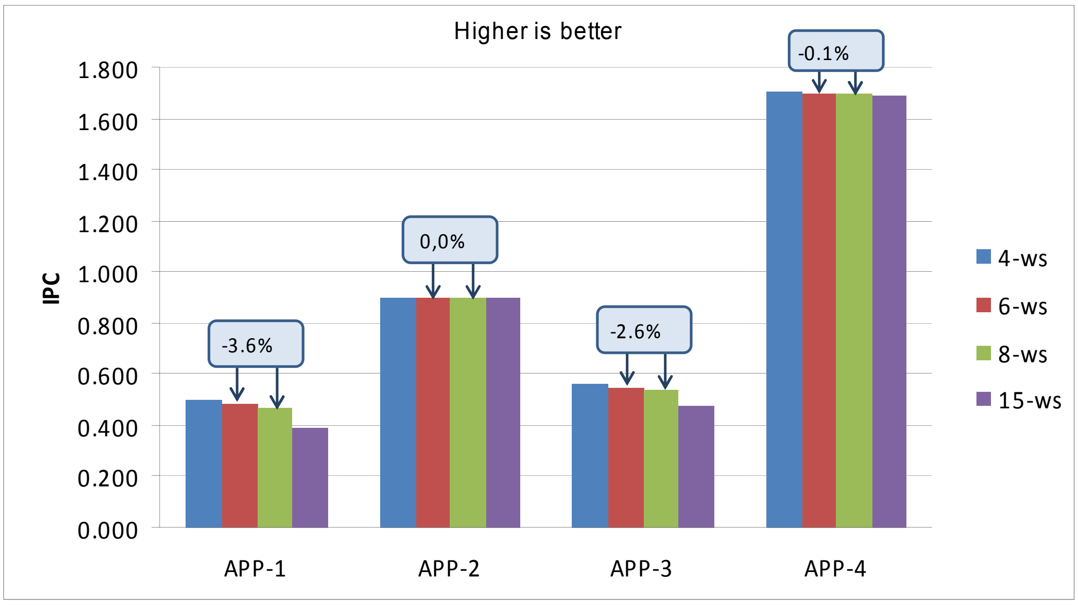


Figure 4 IPC versus PMU0 Program Flash wait states

3.2.2 Caches: ICACHE and DCACHE

16KB of ICACHE and 16KB DCACHE are used to reduce an average access time of much slower but much bigger PMU0/PMU1 Flashes, LMU RAM and external RAM/Flash. The code and data used can be fully cacheable partially or not cacheable depended on application requirements.

3.2.3 Instruction Cache ICACHE

The ICACHE is a four-way set-associative cache with a Pseudo Least-Recently-Used (PLRU) replacement algorithm. Each ICACHE line contains 256 bits of instruction along with a single associated valid bit and associated ECC bits.

Code located in PMU0/PMU1, external RAM/Flash or LMU will be **cacheable** if **all** conditions are fulfilled

- Code located in 0x8 segment (not LMU)
- Code located in 0x9 segment (LMU)
- ICACHE enabled (default, not configurable)
- ICACHE bypass deactivated

Code located in PMU0/PMU1, external RAM/Flash or LMU will be **not cacheable** if **one** of the conditions is fulfilled

- Code located in 0xA segment (not LMU)
- Code located in 0xB segment (LMU)
- ICACHE bypass activated

Note: By default all the segments are defined as described above. Still there is additional register PMA0 that can partially change the cacheability of some segments.

Note: Code can be located in 0x8 segment but still be not cacheable or in case of LMU 0x9 segment.

Note: If some code should be not cacheable (e.g. deterministic behavior) locate it in 0xA segment or in case of LMU 0xB segment.

Code located in following memories can be cacheable:

- PMU0-PFlash and BROM
- PMU1-PFlash
- External memories
- LMU

3.2.4 Data Cache: DCACHE

Four-way set associative cache, Pseudo least recently used (PLRU) replacement algorithm

- Cache line size: 256 bits
- Validity granularity: One valid bit per cache line
- Write-back Cache: Writeback granularity: 256 bits
- Refill mechanism: full cache line refill

Data located in PMU0/PMU1, External RAM/Flash and LMU will be **cacheable** if **all** conditions are fulfilled

- Data located in 0x8 segment (not LMU)
- Data located in 0x9 segment (LMU)
- DCACHE enabled (default, not configurable)
- DCACHE bypass deactivated

Data located in PMU0/PMU1, External RAM/Flash or LMU will be **not cacheable** if **one** of the conditions is fulfilled

- Data located in 0xA segment (not LMU)
- Data located in 0xB segment (LMU)
- DCACHE bypass activated

Note: By default all the segments are defined as described above. Still there is additional register PMA0 that can partially change the cacheability of some segments.

Note: Data can be located in 0x8 segment but still be not cacheable or in case of LMU 0x9 segment.

Note: If some data should be not cacheable (e.g. deterministic behavior) locate it in 0xA segment or in case of LMU 0xB segment.

Data located in following memories can be cacheable:

- PMU0-PFlash, DFlash
- PMU1-PFlash
- External memory
- LMU

3.2.5 ICACHE and DCACHE configuration

After reset DCACHE and ICACHE are enabled but bypassed.

To use ICACHE you need to disable the bypass:

- Set configuration register **PCON0.PCBYB=0** (SFR-R **PCON0)**

To use DCACHE you need to disable the bypass:

- Set configuration register **DCON0.PCBYB=0** (SFR-R **DCON0)**

3.2.6 Evaluating execution time in cacheable architecture

Application code or benchmarks which need to be optimized and the achieved execution speed measured often include a small selected part of application or in special cases a complete code. During optimization process reducing of execution time is one of most important targets. The progress can be evaluated based on relative performance compared to other setting measured under the same conditions.

In case the absolute (not relative) execution times are important, the caches impact under different test condition is noticeable and the results interpretation is not straight forward. By running the same application more than one time (assume the same code and conditions are met) it can be observed different run time values for the first and following tests. Different behavior can be also observed for small code size fully matching in cache and big application causing cache swapping.

In case of small applications, first execution start with empty cache while in following tests the complete code and data are in caches, executing as fast as from scratch pad rams. The run time differences are dependent on the code data structure as seen in **Figure 5** APP-1 and APP-2. Small differences are hints to insensitivity to cache swapping and flashes wait states using cacheable segments. Mapping of the first/second execution time results to the real execution environment is not straightforward.

In case of big application, the cache swapping takes place already in the first and the following tests. The execution time of the second run should be similar to real execution environment.

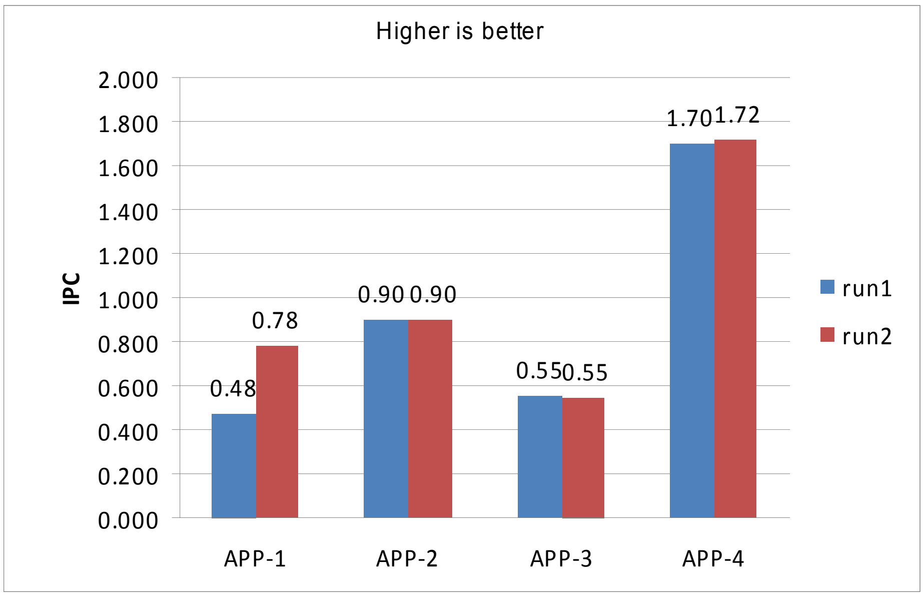


Figure 5 IPC dependency on ICACHE and DCACHE states (run1, run2)

4 Application Software

4.1 Compiler Optimizations

The most significant opportunity for influencing the performance of a given application is by compiler and linker optimizations. Optimizing is a tradeoff between code size and performance. Code optimization is primary controlled by set of optimization flags which together makes optimization profile. Some proved predefined optimization profiles are usually provided by compiler vendors with additional tradeoff parameter for speed or code size.

Application performance optimization is an iterative process of

- Selecting optimization profile
- Executing
- Result recording of execution time and code size
- Comparing to other result
- Repeating if not satisfied

Two main approaches can be used

- Global Optimization: the same optimization setting is applied to complete code
- Profiling driven selective optimization: based on profiling information the critical functions are identified and the best (custom) optimization is applied.

The focus in this document is on global optimization. Included results are also based on global optimization.

Note: Before you start with compiler optimization you should be aware that many options are predefined (having default values) influencing the performance without explicitly be defined. To see option summary including their default values run the ctc.exe using -? option (***ctc -?)*** *located in Altium TASKING compiler directory*

4.1.1 Predefined compiler configurations

Unless you have your proven preferred configuration you should start the optimization process with available predefined optimization profiles -O0 till –O3.

Table 4 Predefined compiler optimization profiles

optimize[=flags] **-Oflags**	**Description**
--optimize=0 -O0	**No optimization** Alias for -OaCEFGIKLMNOPRSUVWY
--optimize=1 -O1	**Optimize** Alias for -OaCefgIKLMNOPRSUVWy
--optimize=2 -O2	**Optimize more (default)** Alias for -OacefgIklMNoprsUvwy
--optimize=3 -O3	**Optimize most** Alias for -OacefgiklmNoprsuvwy

The predefined optimizations profiles including 18 compiler optimization flags (small letter means active) building proven recommended configuration. You can see the differences between various settings by comparing the different letters. For example –O3 has two additional optimization activated m and u. You can use your own setting by explicitly defining all options (letters) or make small modification to available configuration e.g. to use -O2 but disable code compaction (r/R) use –O2 –OR.

Table 5 C Compiler Flags definitions

--optimize[=*flags*]	**-O**[=*flags*]	**Description**
+/-coalesce	a/A	Coalescer: remove unnecessary moves
+/-cse	c/C	Common sub expression elimination
+/-expression	e/E	Expression simplification
+/-flow	f/F	Control flow simplification
+/-glo	g/G	Generic assembly code optimizations
+/-inline	i/I	Automatic function inlining
+/-schedule	k/K	Instruction scheduler
+/-loop	l/L	Loop transformations
+/-simd	m/M	Perform SIMD optimizations
+/-align-loop	n/N	Align loop bodies
+/-forward	o/O	Forward store
+/-propagate	p/P	Constant propagation
+/-compact	r/R	Code compaction (reverse inlining)
+/-subscript	s/S	Subscript strength reduction
+/-unroll	u/U	Unroll small loops
+/-ifconvert	v/V	Convert IF statements using predicates
+/-pipeline	w/W	Software pipelining
+/-peephole	y/Y	Peephole optimizations

Optimization profiles are always extended and influenced by **–tradeoff** parameter controlling the balance of speed versus code size.

4.1.2 Optimizations Tradeoffs: speed vs. code size

Important part of the optimization is finding appropriate tradeoff between code size and performance. Using trade of parameter **--tradeoff**={0|1|2|3|4} or **-t**{0|1|2|3|4} with default: **--tradeoff**=4 optimize for size.

--tradeoff=0: optimize for speed

--tradeoff=4: optimize for code size

--tradeoff=2: balance for speed and size

If the compiler uses certain optimizations (e.g. **–O2**), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

If you have not specified the option **–optimize** (or **–O**), the compiler uses the default *Optimize more* optimization (**-O2**). In this case it is still useful to specify a trade-off level.

It is recommended to start with -**O2** and **--tradeoff=2** setting, analyzing the speed and code size variation as you modifying the Optimization (-O) and --tradeoff settings using your target Application.

Additionally to compiler optimization settings, using of efficient addressing modes can improve significantly overall performance and code size as described in next paragraphs.

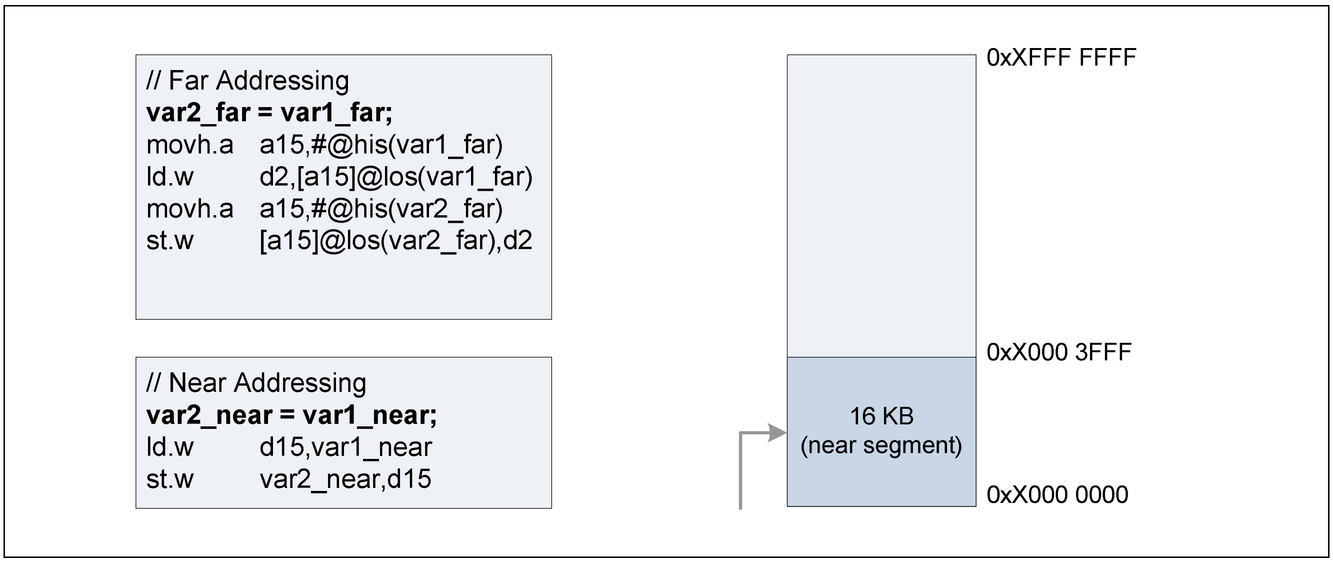
4.1.3 Short addressing

TriCore Architecture has an address width of 32 bit and can access up to 4 GB of memory. Within limited addressing ranges more efficient code and faster execution time can be achieved. Absolute addressing is available for the first 16 KB of each segment. Base + Long Offset addressing using global Base Registers (A0, A1, A8, A9) provide efficient data access in the address range of 64KB. Appropriate tool-chain settings are required to use these memory segments.

4.1.3.1 Near Addressing

Figure 6 shows the location of near segments occupying first 16kB of each TriCore 256MB memory segment. . You can use it to locate variables (initialized or not) and constants. The included example demonstrates the efficiency of absolute addressing used in near segment. Table 6 shows the memories suitable for near addressing.

Note: Do not block near segment with CSA, Stacks, etc. that are not accessed with near addressing



```
// Far Addressing
var2_far = var1_far;
movh.a    a15,#@his(var1_far)
ld.w      d2,[a15]@los(var1_far)
movh.a    a15,#@his(var2_far)
st.w      [a15]@los(var2_far),d2
```

```
// Near Addressing
var2_near = var1_near;
ld.w      d15,var1_near
st.w      var2_near,d15
```

Figure 6 Near versus far Addressing

Table 6 Memories suitable for near data addressing (memories including 0..0x3FFF range)

Memory	Data types	Compiler default Section
PMU0-PFlash	constant	.zrodata
DSPR	variables initialized, uninitialized	.zdata .zbss .nearnoclear
LMU	variables initialized, uninitialized	.zdata .zbss .nearnoclear

4.1.3.2 Configuration for near addressing

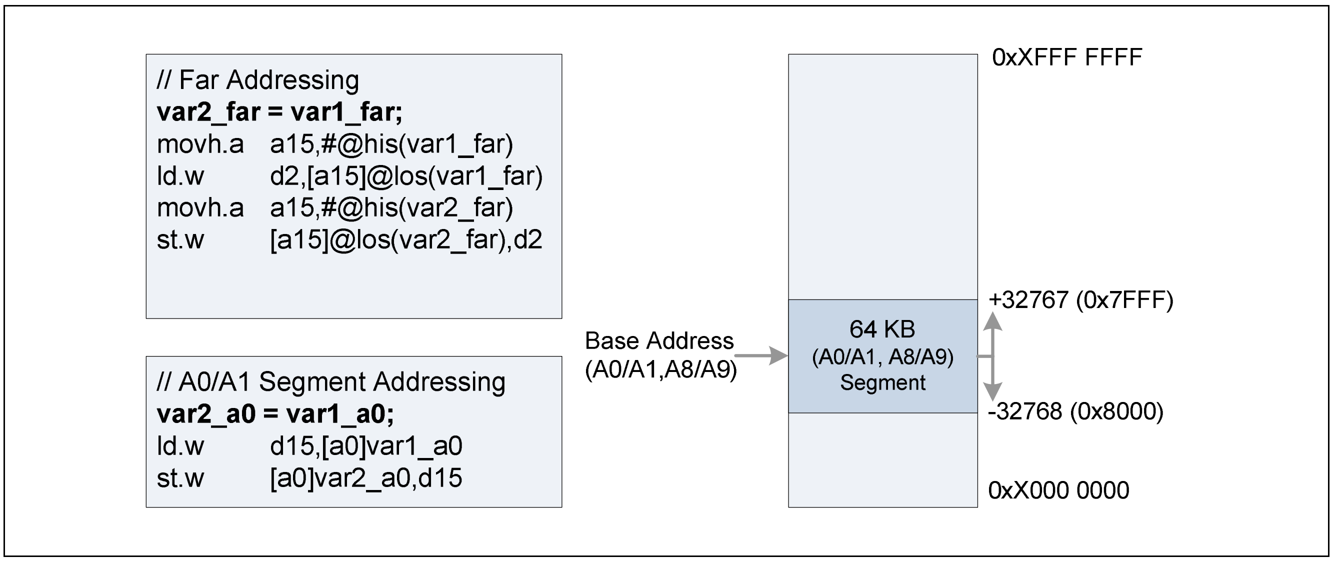
Near addressing can be enabled by

- Compiler option: **--default-near-size** [=threshold]
  (To put all data objects with a size of [threshold] bytes or smaller in __near sections (default threshold =8))
- Pragma in C source code: **default_near_size** [value] [default | restore]
- Memory qualifier: __near

By using compiler option, the setting is valid for the entire program unless not the same options are used for all modules. Pragmas overrules the compiler options and can be used for defined code blocks. With __near memory qualifier you can control the location for single data objects.

4.1.3.3 Base + Long Offset addressing using global Base Registers (A0, A1, A8, A9)

Figure 7 shows the location of A0/A1, A8/A9 addressing range occupying 64kB memory segment. The simple example demonstrates the efficiency of this addressing mode.



```
// Far Addressing
var2_far = var1_far;
movh.a    a15,#@his(var1_far)
ld.w      d2,[a15]@los(var1_far)
movh.a    a15,#@his(var2_far)
st.w      [a15]@los(var2_far),d2
```

```
// A0/A1 Segment Addressing
var2_a0 = var1_a0;
ld.w      d15,[a0]var1_a0
st.w      [a0]var2_a0,d15
```

Figure 7 A0/A1 versus far addressing

A0 register is only available for variables (initialized or not). A1 register is only available for constants. In case of A8 and A9 both are available for variables (initialized or not) and constants.

Table 7 Memories suitable for A0/A1 addressing

Memory	Data types	Compiler default Section
PMU0-PFlash	Constant (**A0**)	.ldata
DSPR*	variables initialized, uninitialized (**A1**)	.sdata .sbss
LMU	variables initialized, uninitialized (**A1**)	.sdata .sbss
Ext. RAM	variables initialized, uninitialized (**A1**)	.sdata .sbss
Ext. FLASH	Constant (**A0**)	.ldata

Table 8 Memories suitable for A8/A9 addressing

Memory	Data types	Compiler default Section
PMU0-PFlash	constant	.rodata_a8 or .rodata_a9
DSPR*	variables initialized, uninitialized	.data_a8 .bss_a8 or .data_a9 .bss_a9
LMU	variables initialized, uninitialized	.data_a8, .bss_a8 or .data_a9 .bss_a9
Ext. RAM	variables initialized, uninitialized	.data_a8 .bss_a8 or .data_a8 .bss_a9
Ext. FLASH	constant	.rodata_a8 or .rodata_a9

4.1.3.4 Configuration for A0 / A1 and A8 / A9 addressing

A0/A1 addressing can be enabled by

- Compiler option: **--default-a0-size** [=threshold]
  (To put all data objects with a size of [threshold] bytes or smaller in A0 section (default threshold =0))
- Compiler option: **--default-a1-size** [=threshold]
  (To put all data objects with a size of [threshold] bytes or smaller in A1 section (default threshold =0))

- Pragma in C source code: **default_a0_size** [value] [default | restore]
- Pragma in C source code: **default_a1_size** [value] [default | restore]
- Memory qualifier: __a0, __a1

A8/A9 addressing can be enabled by

- Memory qualifier: __a8, __a9

By using compiler option, the settings are valid for the entire program unless not the same options are used for all modules. Pragmas overrules the compiler options and can be used for defined code blocks. With __a0, __a1, __a8, __a9 memory qualifier you can control the location for single data objects.

Note: A8/A9 location options doesn't include default-a8/a9-size with threshold as in A0/A1 (e.g. ***--default-a1-size)***

4.1.3.5 Qualifiers controlling all addressing modes

Following qualifiers controlling all addressing modes

- **for_constant_data_use_memory** *memory*
- **for_extern_data_use_memory** *memory*
- **for_initialized_data_use_memory** *memory*
- **for_uninitialized_data_use_memory** *memory*

Use the specified memory for the type of data mentioned in the pragma name. You can specify the following memories: **near, far, a0, a8** or **a9**. For pragma for_constant_data_use_memory you can also specify the **a1** memory.

4.1.4 Memory Location (Typical Use Case)

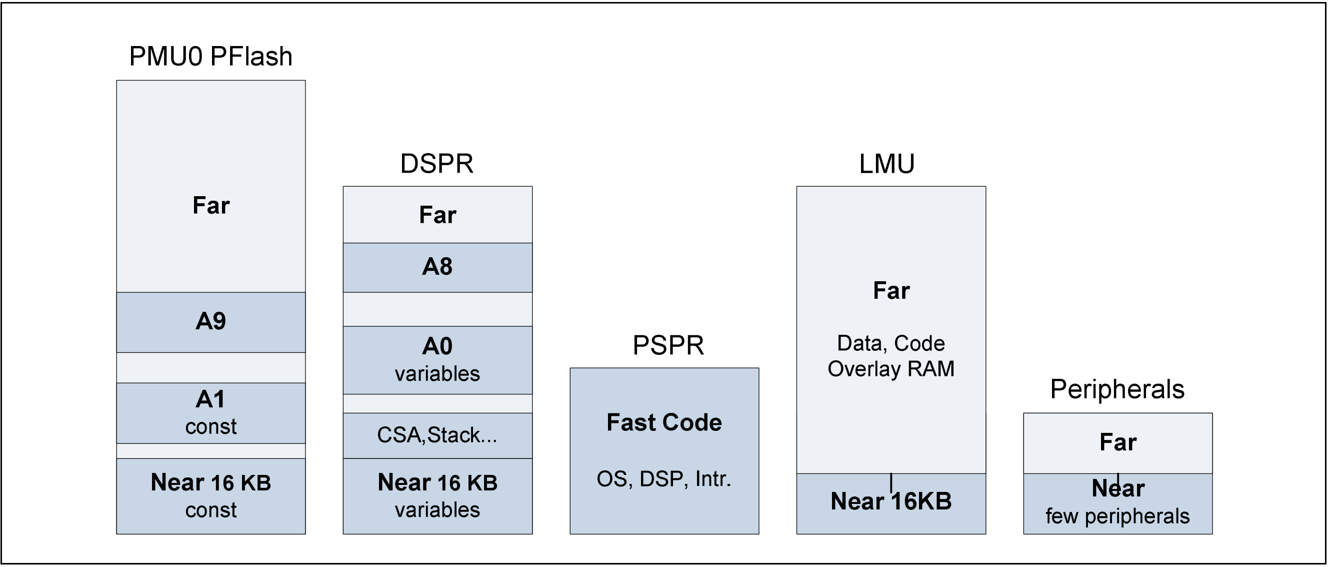


Figure 8 Memory locations (Typical Use Case)

4.2 Linker Script files

TriCore memory system with its hierarchy of different memories and caches plays important role in overall system performance. Controlling the location of data and code whether it should be cacheable or not in fast or slow memories is primary resolved by linker scripts files. Altium TASKING provides predefined linker script file for each device stored in *.lsl files

4.3 Additional optimization options

You can further improve the performance of your software if following measures are suitable to your application.

4.3.1 Floating-point arithmetic/algorithms

FPU: TriCore architecture includes high performance IEEE-754 compliant single-precision Floating Point Unit. Set the compiler option **–fpu-present** to use the FPU instead of emulation library.

Double as float: In some cases double precision floating point data is included in the source code (e.g. model based automatic code generation) but single precision can be used. By setting **--no-double option** the compiler treats variables of the type double as float.

Exception handling: Hardware implemented FPU exception monitoring with appropriate trapping functionality releasing the application from continuous monitoring in software

4.3.2 Fixed-point arithmetic/algorithms

TriLib - TriCore DSP Library: Hand-coded assembly implemented C-callable highly optimized library of common DSP algorithms using fixed-point arithmetic.

4.3.3 Intrinsic Functions

Some specific assembly instructions have no equivalence in C. Intrinsic functions give the possibility to use these instructions. Intrinsic functions are predefined functions that are recognized by the compiler which always inlines the corresponding assembly instructions in the assembly source (rather than calling it as a function).

Table 9 Intrinsic Functions overview

Intrinsic functions groups	Example	Description
Minimum and maximum of (short) integers	int __min(int, int)	Return minimum of two integers
Fractional data type support	__sfract __round16(__fract)	Convert __fract to __sfract
Packed data type support	char __extractbyte1(__packb)	Extract first byte from a __packb
Interrupt handling	void __enable (void)	Enable interrupts immediately at function entry
Insert single assembly instruction	void __nop(void)	Insert NOP instruction
Register handling	int __clz (int)	Count leading zeros in int
Insert / extract bit-fields and bits	void __putbit(int value, int* address, int bitoffset)	Store a single bit

4.3.4 Inline assembler

Using inline assembly you can use assembly instructions in the C source and pass C variables as operands to the assembly code. Primary usable for small and efficient code sequences or target specific operations not available in ANSI-C.

5 Performance Optimization Checklist

Table 10 Hardware Configuration

Description	Details	Default (after reset)
CPU Clock	Check the CPU Clock frequency See **3.1**	Free running mode (~17 MHz)
ICACHE	Verify the ICACHE is usable = bypass is disabled See **3.2.5**	Enabled but bypassed
DCACHE	Verify the DCACHE is usable = bypass is disabled See **3.2.5**	Enabled but bypassed
PMU0 FLASH0, PMU1 FLASH1	Check the wait state setting for PFLASH and DFLASH see **3.2.1.1**	PFLASH 8 wait states DFLASH 15 wait states
Performance Counters	To be used need first be enabled see **Note:**	Disabled

Table 11 Software Configuration

Description	Details	Default
Compiler configuration	To check used compiler configuration see *.src generated files.	You can see the defaults settings by running **ctc -?**
Linker configurations and location	Check the MAP file to control used memories, segments and data/code location	-
Using of cacheable segments	To use caches, additionally to HW configuration the code/data need to be located in cacheable segment see **3.2.3** and **3.2.4**	-
Using FPU	FPU should be used instead of emulation library. To verify check MAP file, should include libc_fpu.a or libcs_fpu.a and libfp_fpu.a. See **4.3.1** and **Table 17**	Derived from --cpu option or explicitly defined by **--fpu-present**
Double as Single	If applicable, replace double with single precision. libcs_fpu.a instead of libc_fpu.a will be used. See **4.3.1** and **Table 17**	Not set.

6 Performance relevant differences TC1.6 vs. TC1.3.1

TriCore 1.6 architecture includes many hardware improvements having impact on the application performance but not visible for software developer. Still there are changes as longer pipeline, new instruction or extended instruction functionality having impact on performance and should be considered in optimization process.

6.1 Pipeline

Longer pipeline of TC1.6 has impact on optimal coding for dual issue instruction involving the Load/Store and Integer pipeline. It is primary relevant for hand-coded assembly implementations. In case of C- language the compiler is already adapted to handle this change.

6.2 Base + Long Offset addressing

TC1.3.1 Load/Store instructions using Base + Long offset addressing is limited to Word data type.

TC1.6 supports Word, Halfword and Byte data types. This addressing mode, used for A0/A1 and A8/A9 addressing segments, can be used for all three data types over full segment range of 64 KB.

6.3 Hardware Floating Point Unit (FPU)

TC1.6 implements a fully pipelined FPU working in parallel with the existing integer pipeline. Overall higher floating point performance is expected.

6.4 Integer Division

TC1.3.1 32bit/32bit division ~20cyc

```
dvinit   e8,d4,d0
dvstep   e8,e8,d0
dvstep   e8,e8,d0
dvstep   e8,e8,d0
dvstep   e8,e8,d0
dvadj    e8,e8,d0
```

TC1.6 new instructions DIV & DIV.U have been implemented executing in ~9 cycles.

6.5 Performance TC1.6 vs. TC1.3.1 architecture

Figure 9 and Figure 10 shows the performance comparison of both architectures using the same test conditions. Usually the new TC-1.6 architecture is faster and generated smaller code size.

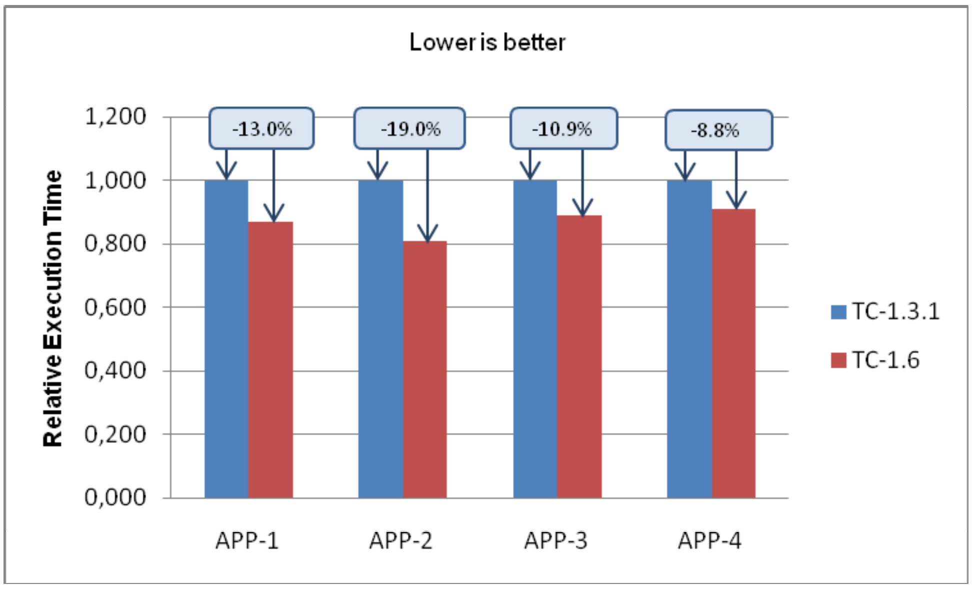


Figure 9 Execution Time of TC-1.6 vs. TC-1.3.1 (same configuration and CPU Clk)

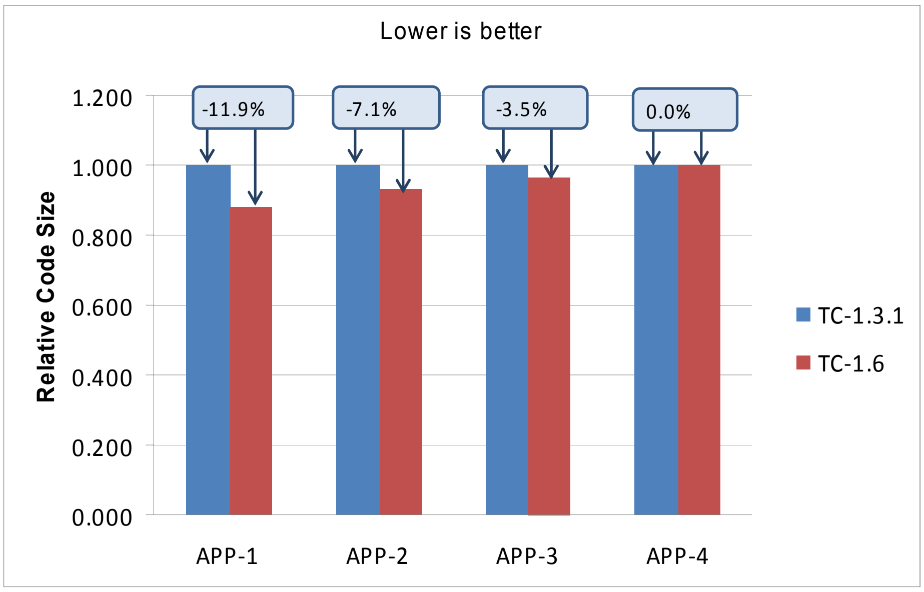


Figure 10 Code Size of TC-1.6 vs. TC-1.3.1 (same configuration and CPU Clk)

7 References

[1] Infineon: TriCore V1.6 User Manual (Vol1)

[2] Infineon: TriCore V1.6 User Manual (Vol2)

[3] Altium TASKING User Manual: ctc_user_guide.pdf

[4] Altium TASKING AppNote: LSL Sample Cases using the Control Program

8 Appendix A

Summary of Altium TASKING configuration options and naming conventions

8.1 Compiler Options

Table 12 Compiler Options (command line syntax)

Long option name **Short option name**	**Default**	**Description**
optimize[=flags] **-O**flags	-O2	C Compiler optimization options. Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a +longflag. To switch a flag off, use an uppercase letter or a -longflag. Separate longflags with commas. See also **Table 13** and **Table 14**.
--tradeoff={0|1|2|3|4} **-t**{0|1|2|3|4}	4	If the compiler uses certain optimizations (option --optimize), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).
--default-near-size [=threshold] **-N**[threshold]	8	With this option you can specify a threshold value for __near allocation. If you do not specify __near or __far in the declaration of an object, the compiler chooses where to place the object. The compiler allocates objects smaller or equal to the threshold in __near sections. Larger objects are allocated in __a0, __a1 or __far sections. If you omit a threshold value, all objects will be allocated __near, including arrays and string constants.
--default-a0-size [=threshold] **-Z**[threshold]	0	Used for data: initialized/uninitialized (a0data,a0bss) Allocation in __a0 memory means that the object is addressed indirectly, using A0 as the base pointer. The total amount of memory that can be addressed this way is 64 KB. With this option you can specify a threshold value for __a0 allocation. If you do not specify a memory qualifier such as __near or __far in the declaration of an object, the compiler chooses where to place the object based on the size of the object. First, the size of the object is checked against the near size threshold, according to the description of the --default-near-size (-N) option. If the size of the object is larger than the near size threshold, but lower or equal to the a0 size threshold, the object is allocated in __a0 memory. Larger objects, arrays and strings will be allocated __far.
--default-a1-size [=threshold] **-Y**[threshold]	0	Same as -a0 but used for constant (a1rom)
--fpu-present	-	With this option the compiler can generate single precision floating-point instructions in the assembly file. If you select a valid target processor (command line option --cpu (-C)), this option is automatically set, based on the

		chosen target processor.
--core=*core*	derived from --cpu, if used, otherwise tc1.3	With --core=tc1.6, the compiler can generate TriCore 1.6 instructions in the assembly file. If you select a valid target processor (command line option --cpu (-C)), the core is automatically set, based on the chosen target processor
--cpu=*cpu* -**C**cpu	-	With this option you define the target processor for which you create your application. Based on this option the compiler always includes the special function register file regcpu.sfr, unless you disable the option Automatic inclusion of '.sfr' file on the Preprocessing page (option--no-tasking-sfr). Based on the target processor the compiler automatically detects whether a FPU-unit is present and whether the architecture is a TriCore1.6. This means you do not have to specify the compiler options --fpu-present and --core=tc1.6 explicitly when one of the supported derivatives is selected.
--switch=auto	auto	You can give one of the following arguments: **auto** Choose most optimal code **jumptab** Generate jump tables **linear** Use linear jump chain code **lookup** Generate lookup tables
--align=*value*	0	By default the C compiler aligns objects to the minimum alignment required by the architecture. With this option you can increase this alignment for objects of four bytes or larger. The value must be a power of two.
--inline-max-size=*threshold*	-1	With the option --inline-max-size you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines all functions that are smaller than the specified threshold. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline. The default threshold is -1, which means that the threshold depends on the option --tradeoff.
--inline-max-incr=*percentage*	-1	After the compiler has inlined all functions that have the function qualifier inline and all functions that are smaller than the specified threshold, the compiler looks whether it can inline more functions without increasing the code size too much. With the option --inline-max-incr you can specify how much the code size is allowed to increase. The default value is -1, which means that the value depends on the option --tradeoff.
--immediate-in-code	-	By default the TriCore C compiler creates a data object to represent an immediate value of 32 or 64 bits, then loading this constant value directly into a register. With this option you can tell the compiler to code the immediate values directly into the instructions, thus using less data, but more code. Actually when option --default-near-size < 4, 32-bit immediates will be coded into instructions anyhow, when it is >= 4 they will be located in neardata. When --default-near-size < 8, 64-bit immediates will be located in fardata, when it is >= 8 they will be located in neardata as well.
--compact-max-size=*value*	200	This option is related to the compiler optimization --optimize=+compact (**Code compaction** or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once are

		transformed into a function. This reduces code size (possibly at the cost of execution speed). However, in the process of finding sequences of matching instructions, compile time and compiler memory usage increase quadratically with the number of instructions considered for code compaction. With this option you tell the compiler to limit the number of matching instructions it considers for code compaction.
--max-call-depth= *value*	-1	This option is related to the compiler optimization --optimize=+compact (**Code compaction** or reverse inlining). During code compaction it is possible that the compiler generates nested calls. This may cause the program to run out of its stack. To prevent stack overflow caused by too deeply nested function calls, you can use this option to limit the call depth. This option can have the following values: **-1** Poses no limit to the call depth (default) **0** The compiler will not generate any function calls. (Effectively the same as if you turned off code compaction with option --optimize=-compact) **>0** Code sequences are only reversed if this will not lead to code at a call depth larger than specified with value. Function calls will be placed at a call depth no larger than value-1. (Note that if you specified a value of 1, the option --optimize=+compact may remain without effect when code sequences for reversing contain function calls.)

8.2 Compiler optimizations flags

Table 13 Compiler –optimize (-O) Flags (command line syntax)

--optimize[=*flags*]	**-O***[=flags]*	**Description**
+/-coalesce	a/A	Coalescer: remove unnecessary moves
+/-cse	c/C	Common sub expression elimination
+/-expression	e/E	Expression simplification
+/-flow	f/F	Control flow simplification
+/-glo	g/G	Generic assembly code optimizations
+/-inline	i/I	Automatic function inlining
+/-schedule	k/K	Instruction scheduler
+/-loop	l/L	Loop transformations
+/-simd	m/M	Perform SIMD optimizations
+/-align-loop	n/N	Align loop bodies
+/-forward	o/O	Forward store
+/-propagate	p/P	Constant propagation
+/-compact	r/R	Code compaction (reverse inlining)
+/-subscript	s/S	Subscript strength reduction
+/-unroll	u/U	Unroll small loops
+/-ifconvert	v/V	Convert IF statements using predicates
+/-pipeline	w/W	Software pipelining
+/-peephole	y/Y	Peephole optimizations

8.3 Predefined compiler optimization profiles

Table 14 Predefined compiler optimization profiles

optimize[=flags] **-Oflags**	**Description**
--optimize=0 -O0	**No optimization** Alias for -OaCEFGIKLMNOPRSUVWY
--optimize=1 -O1	**Optimize** Alias for -OaCefgIKLMNOPRSUVWy
--optimize=2 -O2	**Optimize more (default)** Alias for -OacefgIklMNoprsUvwy
--optimize=3 -O3	**Optimize most** Alias for -OacefgiklmNoprsuvwy

8.4 Compiler generated sections

Table 15 Compiler generated sections

Section type	Name prefix	Description
code	.text	program code
neardata	.zdata	initialized __near data
fardata	.data	initialized __far data
nearrom	.zrodata	constant __near data
farrom	.rodata	constant __far data
nearbss	.zbss	uninitialized __near data (cleared)
farbss	.bss	uninitialized __far data (cleared)
nearnoclear	.zbss	uninitialized __near data
farnoclear	.bss	uninitialized __far data
a0data	.sdata	initialized __a0 data
a0bss	.sbss	uninitialized __a0 data (cleared)
a1rom	.ldata	constant __a1 data
a8data	.data_a8	initialized __a8 data
a8rom	.rodata_a8	constant __a8 data
a8bss	.bss_a8	uninitialized __a8 data (cleared)
a9data	.data_a9	initialized __a9 data
a9rom	.rodata_a9	constant __a9 data
a9bss	.bss_a9	uninitialized __a9 data (cleared)

8.5 Compiler memory qualifiers

Table 16 Compiler memory qualifiers

Qualifiers	Description	Location	Maximum object size	Pointer size	Section types
__near	Near data, direct addressable	First 16 kB of a 256 MB block	16 kB	32-bit	neardata, nearrom, nearbss, nearnoclear
__far	Far data, indirect addressable	Anywhere	no limt	32-bit	fardata, farrom, farbss, farnoclear
__a0	Small data	Sign-extended 16-bit offset from address register A0	64 kB	32-bit	a0data, a0bss
__a1	Literal data, read-only	Sign-extended 16-bit offset from address register A1	64 kB	32-bit	a1rom
__a8	Data, reserved for OS	Sign-extended 16-bit offset from address register A8	64 kB	32-bit	a8data, a8rom, a8bss
__a9	Data, reserved for OS	Sign-extended 16-bit offset from address register A9	64 kB	32-bit	a9data, a9rom, a9bss

8.6 Libraries

Table 17 Libraries

Libraries	Description

libc[s].a
libc[s]_fpu.a
	C libraries
	Optional letter:
	s = single precision floating-point (compiler option **--no-double**
	_fpu = with FPU instructions (compiler option **--fpu-present**)

libfp[t].a
libfp[t]_fpu.a
	Floating-point libraries:
	Optional letter
	t = trapping (control program option **--fp-trap**
	_fpu = with FPU instructions (compiler option **--fpu-present**)

librt.a	Run-time library

libpb.a
libpc.a
libpct.a
libpd.a
libpt.a
	Profiling libraries
	pb = block/function counter
	pc = call graph
	pct = call graph and timing
	pd = dummy
	pt = function timing

libcp[s][x].a	C++ libraries
	Optional letter:
	s = single precision floating-point
	x = exception handling

libstl[s]x.a	STLport C++ libraries (exception handling variants only) Optional letter: s = single precision floating-point