Memory_Access_Performance_1 for KIT_AURIX_TC397_TFT

Memory access performance

AURIX™ TC3xx Microcontroller Training
V1.0.0

Scope of work

Data is stored and read back from different RAM locations while the access times are monitored using the CPU performance counters.

A routine is implemented to execute performance measurements using the CPU performance counters. The amount of required CPU clocks is evaluated for ten alternating write and read operations to different memories (DSPR0, DSPR1, LMU).

Introduction

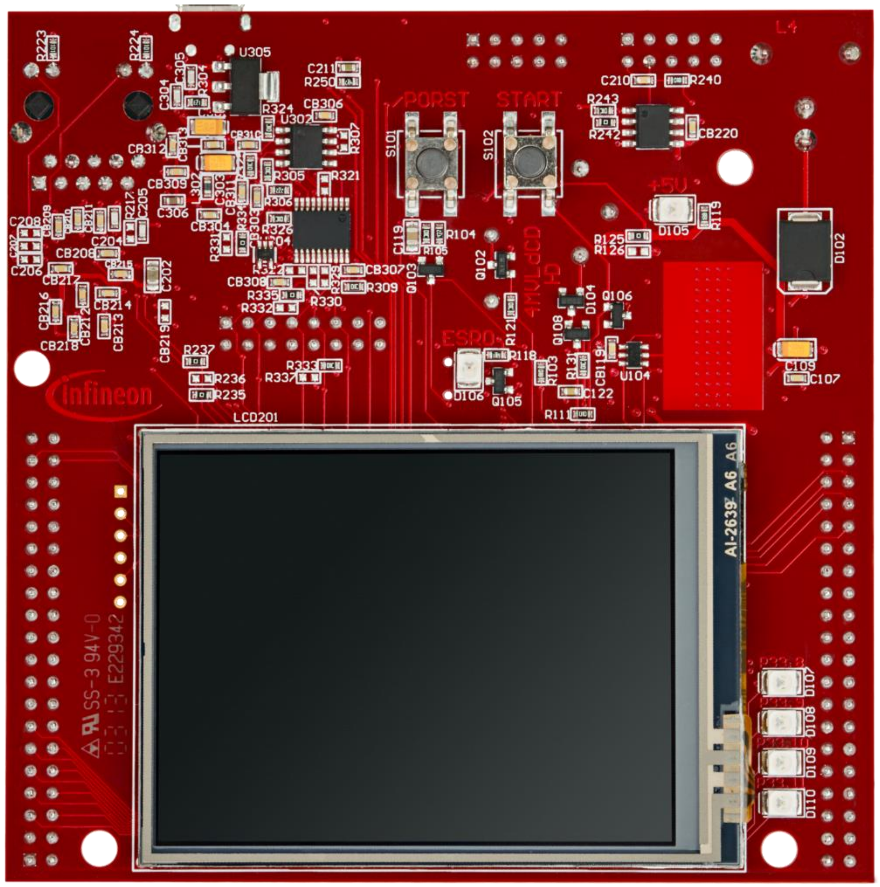
- The microcontrollers of the AURIX™ TC3xx family have several CPU related volatile memories. Some are dedicated for a certain CPU e.g. the Program Scratch Pad RAM (PSPR), the Data Scratch Pad RAM (DSPR), PCACHE and DCACHE and some are used for general purpose such as the Local Memory Unit (LMU), where the access latency is typically higher.

- A higher access latency is also expected if a certain CPU accesses the memory adjacent to another CPU via the SRI bus.

- The latency in CPU clock cycles can be easily measured using the CPU performance counters.

Hardware setup

This code example has been developed for the board KIT_A2G_TC397_5V_TFT.



Implementation

Defining global data fields within the different volatile memories

› The global macro ***BUFFERSIZE*** defines the size of ***10*** elements for the data fields.

› Alternate writes and reads are repeated ten times to determine the number of CPU clocks. The global array ***g_DataLMU[BUFFERSIZE]*** defines the data field in the LMU. Furthermore, the global arrays ***g_DataDSPR0[BUFFERSIZE]*** and ***g_DataDSPR1[BUFFERSIZE]*** define the data fields in the ***DSPR0*** and ***DSPR1***. The compiler directive ***#pragma*** is used to force the compiler to put the global variables in the corresponding volatile memory.

Sequence of memory performance measurements

› A series of measurements is conducted by repeatedly invoking the function ***accessData()*** with different parameters inside the function ***measurePerformance()***. The first argument is the data field where the write/read access goes to/comes from and the second argument is a container to store the results of the performance counter.

Note: For using the performance counters, the On-Chip Debug System has to be enabled. This is ensured by the debugger itself, therefore the example has to be tested within an active debug session.

Implementation

Executing write and read accesses

- The function ***IfxCpu_resetAndStartCounters()*** starts the CPU performance counters. A parameter is specifying the counter mode:
  - **Normal** mode: The counters start counting as soon as they are enabled and will keep counting until they are disabled.
  - **Task** mode: The counters will only count if the processor detected a debug event with the action to start the performance counters.

- A ***for*** loop is used to write the index value into the data field and read it back. The first parameter of the function ***accessData()*** is specified as volatile. This prevents the compiler from optimizing the assembler code in order to ensure that the index of the for loop is read back from the memory.

Implementation

Executing write and read accesses

- To ensure that the Store Buffer is flushed before starting the read operation, a ***DSYNC*** TriCore™ instruction is performed using the __***dsync()*** intrinsic function.
- The function ***IfxCpu_stopCounters()*** stops and returns the counters' values.

Both the functions ***IfxCpu_resetAndStartCounters()*** and ***IfxCpu_stopCounters()*** are part of the Infineon low level drivers and are implemented in ***IfxCpu.h***.

Note: In TriCore™ TC1.6.2P CPUs, store buffering is implemented to decouple memory write operations from CPU instruction execution, thus increasing their performance. All stores from the CPU are placed in the store buffer prior to being written to local memory or transferred via the bus system. Therefore, if there is a requirement that data is written to local memory prior to execution of a subsequent instruction, then a ***DSYNC*** instruction may be used to flush the store buffers. In this simple example, the ***DSYNC*** instruction is needed, otherwise the write-read operations can be "optimized" by the store buffer and the memory access timings will have the same value across different types of memories.

Implementation

Disabling the DCACHE

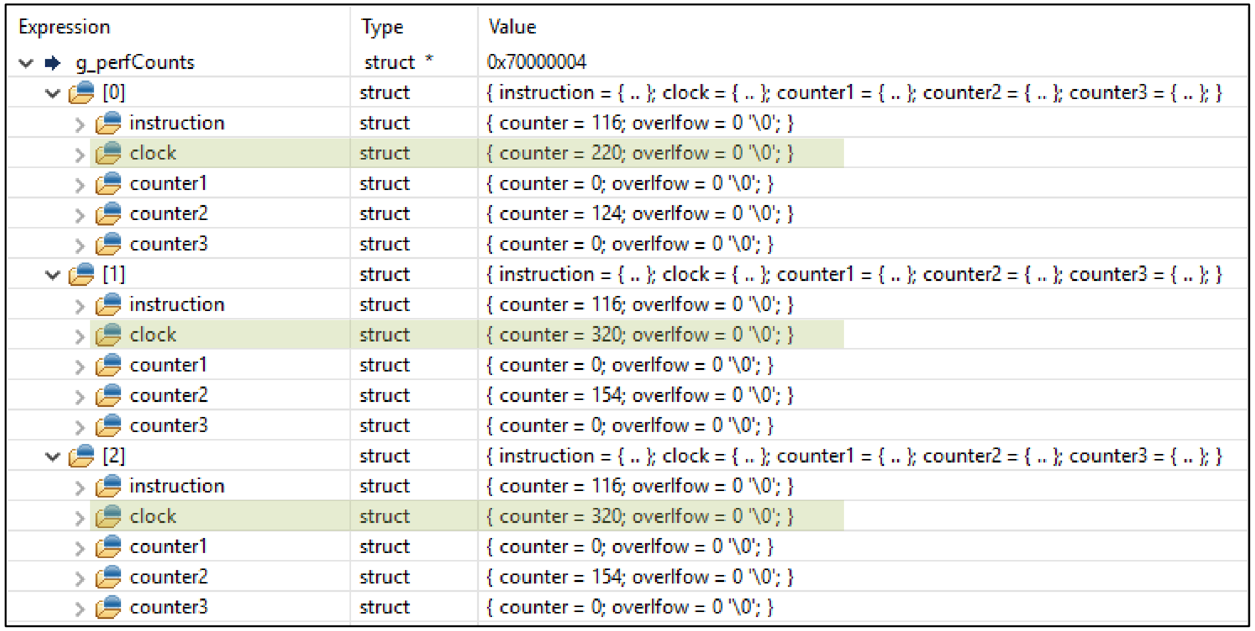
The CPU uses the DCACHE for data storage into the LMU. This is prevented by switching off the DCACHE usage. The macro ***IFX_CFG_SSW_ENABLE_TRICORE0_DCACHE*** is required to be set to zero.

The macro is configured inside the header file ***Ifx_Cfg.h***.

Run & Test

After code compilation and flashing the device, check the amount of clocks representing the access latency in the debugger perspective:

- Add the global variable ***g_perfCounts*** to the Watch window and check the elements 0,1 and 2
- 0: CPU0 writes and reads to/from DSPR0
- 1: CPU0 writes and reads to/from DSPR1
- 2: CPU0 writes and reads to/from LMU



Expression	Type	Value
g_perfCounts	struct *	0x70000004
[0]	struct	{ instruction = { .. }; clock = { .. }; counter1 = { .. }; counter2 = { .. }; counter3 = { .. }; }
instruction	struct	{ counter = 116; overflow = 0 '\0'; }
clock	struct	{ counter = 220; overflow = 0 '\0'; }
counter1	struct	{ counter = 0; overflow = 0 '\0'; }
counter2	struct	{ counter = 124; overflow = 0 '\0'; }
counter3	struct	{ counter = 0; overflow = 0 '\0'; }
[1]	struct	{ instruction = { .. }; clock = { .. }; counter1 = { .. }; counter2 = { .. }; counter3 = { .. }; }
instruction	struct	{ counter = 116; overflow = 0 '\0'; }
clock	struct	{ counter = 320; overflow = 0 '\0'; }
counter1	struct	{ counter = 0; overflow = 0 '\0'; }
counter2	struct	{ counter = 154; overflow = 0 '\0'; }
counter3	struct	{ counter = 0; overflow = 0 '\0'; }
[2]	struct	{ instruction = { .. }; clock = { .. }; counter1 = { .. }; counter2 = { .. }; counter3 = { .. }; }
instruction	struct	{ counter = 116; overflow = 0 '\0'; }
clock	struct	{ counter = 320; overflow = 0 '\0'; }
counter1	struct	{ counter = 0; overflow = 0 '\0'; }
counter2	struct	{ counter = 154; overflow = 0 '\0'; }
counter3	struct	{ counter = 0; overflow = 0 '\0'; }

References

- AURIX™ Development Studio is available online:
- https://www.infineon.com/aurixdevelopmentstudio
- Use the „*Import...*“ function to get access to more code examples.

- More code examples can be found on the GIT repository:
- https://github.com/Infineon/AURIX_code_examples

- For additional trainings, visit our webpage:
- https://www.infineon.com/aurix-expert-training

- For questions and support, use the AURIX™ Forum:
- https://www.infineonforums.com/forums/13-Aurix-Forum

Edition 2020-12
Published by
Infineon Technologies AG
81726 Munich, Germany

Do you have a question about this document?
Email: erratum@infineon.com

Document reference
Memory_Access_Performance_1 _KIT_TC397_TFT