Get started with Arm Cortex-M55

AN 329, Spring 2020, V 1.0

keil-feedback@arm.com

Abstract

This application note contains the project that is used in the webinar "How to get started with Arm Cortex-M55 software development". It explains the project is explained in-depth and repeats the steps that were shown.

The project itself contains four different implementations of a multiply-accumulate (MLA) function. It is explained how the implementations differ and what performance gains you can expect when using optimized code on Arm Cortex-M55. The project can run on a Fixed Virtual Platform (FVP) model that is shipped with MDK v5.30, requiring an MDK-Professional license.

Contents

Introduction

This application note explains how to implement four different versions of a multiply-accumulate function on Arm Cortex-M55. You will learn how to use the performance monitoring unit (PMU) to examine the differences in computing performance. Also, some debugging concepts of MDK are discussed and used.

Finally, the application note quickly touches on the usage of MDK with the Arm MPS3 prototyping board that can host the netlist of Arm Cortex-M55 and that enables real code profiling using target hardware.

Prerequisites

To run the project, you need to install the following software:

- Install MDK v5.30 from www.keil.com/demo/eval/arm.htm
- Add an MDK-Professional license. If you do not have access to this MDK edition, you can request a 30-day trial license from within the tool: www.keil.com/support/man/docs/license/license_eval.htm

Project Structure

The project is configured for the Arm Cortex-M55 and has basically two source files:

- `main.c` contains the main() function and calls the different implementations of the MLA function.
- `mla_functions.S` is an assembly file that contains the different MLA implementations

The MLA implementations are as follows:

- Scalar only MLA function implementation.
- Scalar MLA function implementation with low-overhead loops (LOL) (refer to Appendix).
- Vectorized MLA function implementation with scalar code for loop tail prediction (refer to Appendix).
- Vectorized MLA function implementation with low-overhead loops (refer to Appendix).

The project also contains a custom scatter file (`ARMC55_ac6.sct`) that is used to place one of the software components in an uninitialized part of the target's memory.

Software Components

Apart from the two source files, the project contains the following software components:

- **::CMSIS:Core** for access to the Cortex-M55 header file
- **::Device:Startup** for startup and systems files
- **::Compiler:Event Recorder** and **::Compiler:I/O:STDOUT:EVR** for [retargeting the `printf()` output](retargeting the printf() output) to the **Debug (printf) Viewer** window

main.c

In main.c, we first include a couple of required header files. The EventRecorder.h file is only included if the component is present (which is noted in RTE_Components.h). This will be used in the last step when we remove the component and thus don't want to include its header file:

```
#include "RTE_Components.h"
#include CMSIS_device_header
#ifdef RTE_Compiler_EventRecorder
  #include "EventRecorder.h" // Keil.ARM Compiler::Compiler:Event Recorder
#endif
#include <stdio.h>
```

After that, we create variables that will be used to compute the results based on the `#defines` as shown below

```
#define LENGTH 127

#ifndef DATATYPE
extern int mla_sca(int a[], int b[], int n);
extern int mla_sca_lol(int a[], int b[], int n);
extern int mla_vec(int a[], int b[], int n);
extern int mla_vec_lol(int a[], int b[], int n);

int a[LENGTH];
int b[LENGTH];

#else

extern int mla_sca(short a[], short b[], short n);
extern int mla_sca_lol(short a[], short b[], short n);
extern int mla_vec(short a[], short b[], short n);
extern int mla_vec_lol(short a[], short b[], short n);

short a[LENGTH];
short b[LENGTH];

#endif

volatile uint32_t scalar_cycle_count        = 0;
volatile uint32_t scalar_lol_cycle_count    = 0;
volatile uint32_t vector_scalar_cycle_count = 0;
volatile uint32_t vector_lol_cycle_count    = 0;
```

Next, there is an initialization function that is used to fill the variables with data that will be used for the MLA function:

```
__attribute__((noinline)) void init_arrays(void) {
  int i;

  for (i = 1; i < (LENGTH + 1); i++) {
    a[i - 1] = i;
    b[i - 1] = i;
  }
}
```

In main(), (based on availability) we initialize Event Recorder, enable the PMU and its cycle counter. Then we initialize the array and start doing the calculations with four different implementations. These calculations are enclosed in calls to the cycle counter to retrieve the required information to measure the application performance. The value is then stored in a variable that is printed on the debug console:

```
int main(void) {
#ifdef RTE_Compiler_EventRecorder
  EventRecorderInitialize(EventRecordAll, 1);
#endif
  ARM_PMU_Enable();
  ARM_PMU_CNTR_Enable(PMU_CNTENSET_CCNTR_ENABLE_Msk);
  init_arrays();
  cycle_count_before = ARM_PMU_Get_CCNTR();
  sca_result = mla_sca(a, b, LENGTH);
  cycle_count_after = ARM_PMU_Get_CCNTR();
  scalar_cycle_count = cycle_count_after - cycle_count_before;
#ifdef RTE_Compiler_EventRecorder
  printf("Scalar only          : Result = %d, Cycles = %d\n", sca_result, scalar_cycle_
count);
#endif
```

mla_functions.S

The capital "S" in the file extension indicates to the Arm assembler that the file requires preprocessing as it contains a couple of `#ifndefs` that are used to steer the selected implementation. To see the best result of each implementation, you need to set the following `#defines` in the **Options for Target – C/C++ (AC6) and Asm** tabs:

Result of	C/C++ (AC6) tab	Asm tab
Scalar only		PLAIN
Scalar LOL		SCALOL
Vector LOL		VECLOL
Optimized types	DATATYPE	DATATYPE

The last one uses different data types than the others (also in `main.c`). Instead of 32-bit integers, 16-bit fixed-point values are used that enable higher throughput in a single iteration.

Running the project

This section explains a couple of ways to execute the application in simulation or on real hardware. It also shows how to use MDK to debug the application and how to optimize the debug experience.

The project ZIP file is available for download on www.keil.com/appnotes/docs/apnt_329.asp. Download the ZIP file, unzip it, and double-click the Cortex-M55.uvprojx file to open it in µVision.

Targets

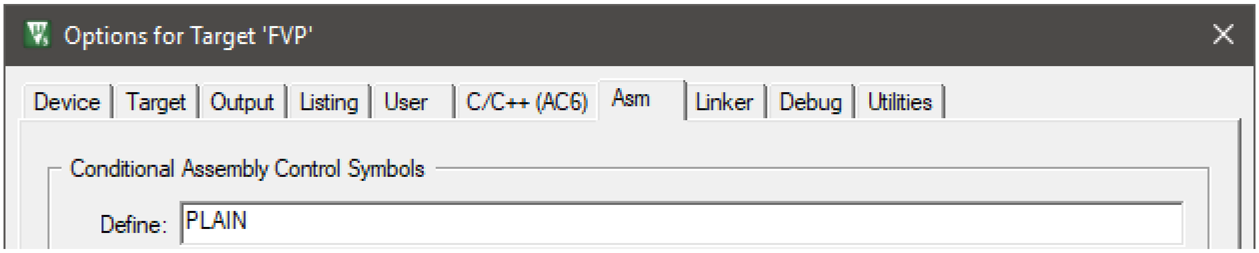
The project supports two targets:

- **FVP**: The project runs in simulation using the Cortex-M55 FVP that is delivered with MDK. An FVP is useful for prototyping as it gives an indication about code performance. However, it does not give accurate measurements.
- **MPS3**: The project connects to a Cortex-M55 FPGA image running on the MPS3 prototyping platform. Using a hardware implementation gives you accurate code performance measurements.

Note: General information on how to use Fixed Virtual Platforms in MDK can be found here: www.keil.com/support/man/docs/fstmdls/

FVP – Scalar only implementation

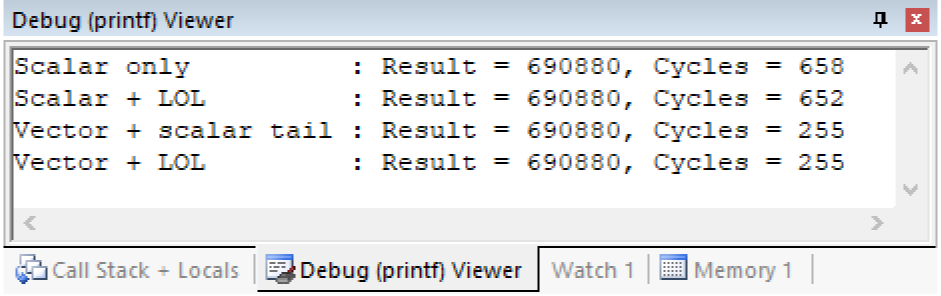
First, we need to tell the assembler to use the right implementation. Go to **Options for Target – Asm** (Atl+F7) and enter `PLAIN` in the **Define:** section:



Define: PLAIN

Build (F7) the project and **Start a Debug Session** (Ctrl + F5). You will see two windows opening in the background – these are issued by the Fast Model and must not be closed during the debug session.

Run (F5) the application. It will hit a breakpoint at the end on the `while(1)` loop. Observe the output of the `printf()` calls in the **Debug (printf) Viewer** window:

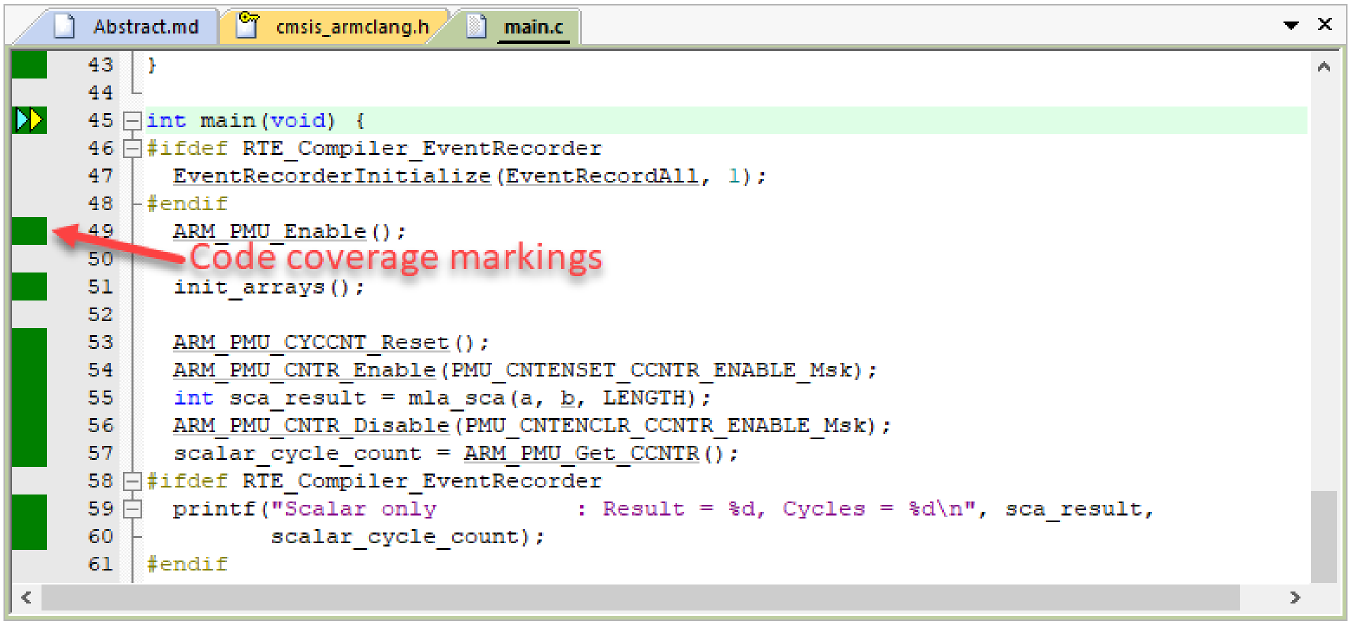


```
Scalar only          : Result = 690880, Cycles = 658
Scalar + LOL         : Result = 690880, Cycles = 652
Vector + scalar tail : Result = 690880, Cycles = 255
Vector + LOL         : Result = 690880, Cycles = 255
```

Note that the cycles do not have to match to your output, but the general relation should be correct.

Code Coverage

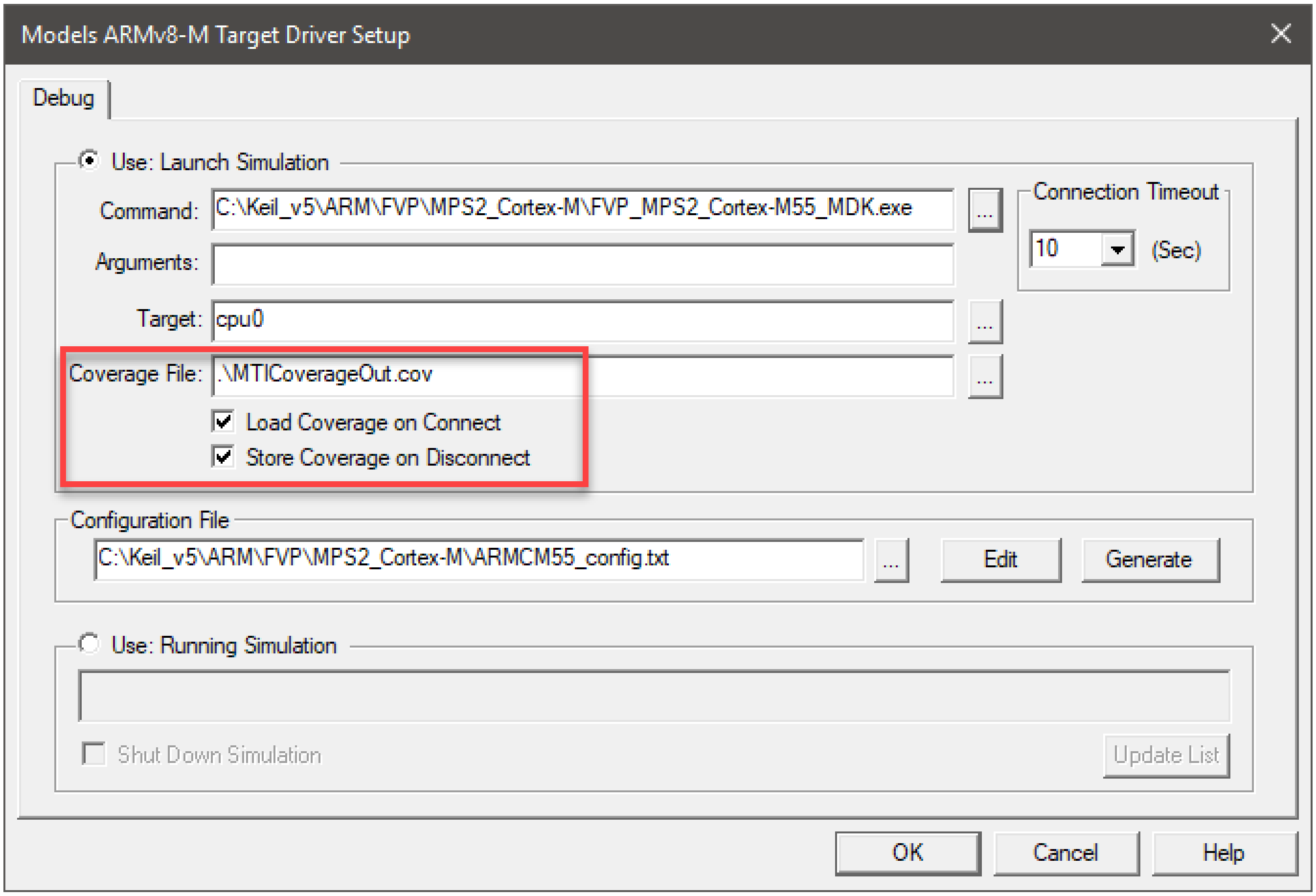
Stop the Debug Session (Ctrl + F5) and restart it immediately afterwards. You will notice that the display of your code has changed. It now contains code coverage markings:



```
}

int main(void) {
#ifdef RTE_Compiler_EventRecorder
  EventRecorderInitialize(EventRecordAll, 1);
#endif
  ARM_PMU_Enable();

  init_arrays();

  ARM_PMU_CYCCNT_Reset();
  ARM_PMU_CNTR_Enable(PMU_CNTENSET_CCNTR_ENABLE_Msk);
  int sca_result = mla_sca(a, b, LENGTH);
  ARM_PMU_CNTR_Disable(PMU_CNTENCLR_CCNTR_ENABLE_Msk);
  scalar_cycle_count = ARM_PMU_Get_CCNTR();
#ifdef RTE_Compiler_EventRecorder
  printf("Scalar only          : Result = %d, Cycles = %d\n", sca_result,
         scalar_cycle_count);
#endif
```

This is due to a new feature in MDK v5.30 that allows to extract coverage information from an FVP. Unfortunately, this cannot be shown live in a debug session, but needs to be loaded when entering debug.

In the **Models ARMv8-M Target Driver Setup** dialog, you can specify to save the coverage information and to load a recorded coverage info on debug entry (**Options for Target – Debug – Settings** (Alt+F7)):



Models ARMv8-M Target Driver Setup

Debug

● Use: Launch Simulation

Command: C:\Keil_v5\ARM\FVP\MPS2_Cortex-M\FVP_MPS2_Cortex-M55_MDK.exe

Arguments:

Target: cpu0

Coverage File: .\MTICoverageOut.cov

☑ Load Coverage on Connect

☑ Store Coverage on Disconnect

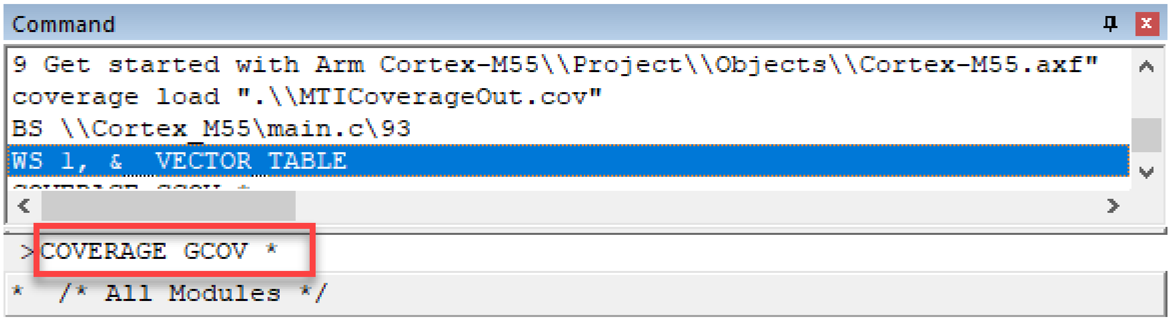
Connection Timeout: 10 (Sec)

Configuration File: C:\Keil_v5\ARM\FVP\MPS2_Cortex-M\ARMCM55_config.txt

○ Use: Running Simulation

□ Shut Down Simulation

In a debug session with the data loaded from the previous run, you can use the `COVERAGE` command to store the coverage information in Gcov format. This is useful for CI/CD environments where your server can run automated testing and create coverage information based on GCOV. Enter the following in the **Command** window:



Command

```
9 Get started with Arm Cortex-M55\\Project\\Objects\\Cortex-M55.axf"
coverage load ".\\MTICoverageOut.cov"
BS \\Cortex_M55\main.c\93
WS 1, & VECTOR_TABLE
```

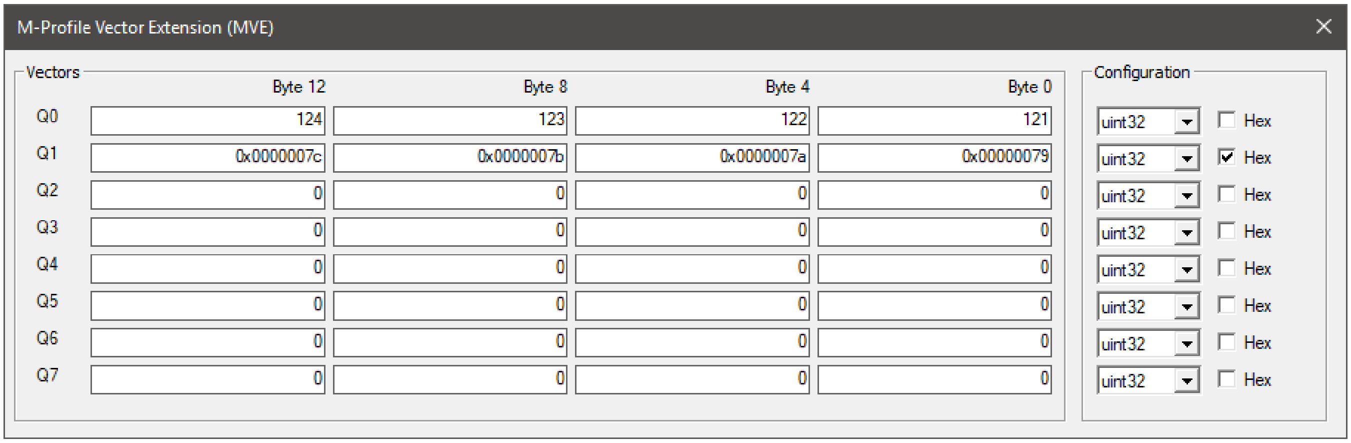
```
>COVERAGE GCOV *
*  /* All Modules */
```

The Gcov files (one for each module) will be saved in the directory where the objects are stored. You can use a tool like gcovr to create a HTML table showing the project's overall code coverage.

M-Profile Vector Extension window

MDK v5.30 introduces a new System Viewer window – the [M-Profile Vector Extension](#) window. This window allows you to check the MVE vector registers.

Go to **View – System Analyzer – Core Peripherals – M-Profile Vector Extension (MVE)** to open the dialog:



M-Profile Vector Extension (MVE)

Vectors

	Byte 12	Byte 8	Byte 4	Byte 0
Q0	124	123	122	121
Q1	0x0000007c	0x0000007b	0x0000007a	0x00000079
Q2	0	0	0	0
Q3	0	0	0	0
Q4	0	0	0	0
Q5	0	0	0	0
Q6	0	0	0	0
Q7	0	0	0	0

Configuration

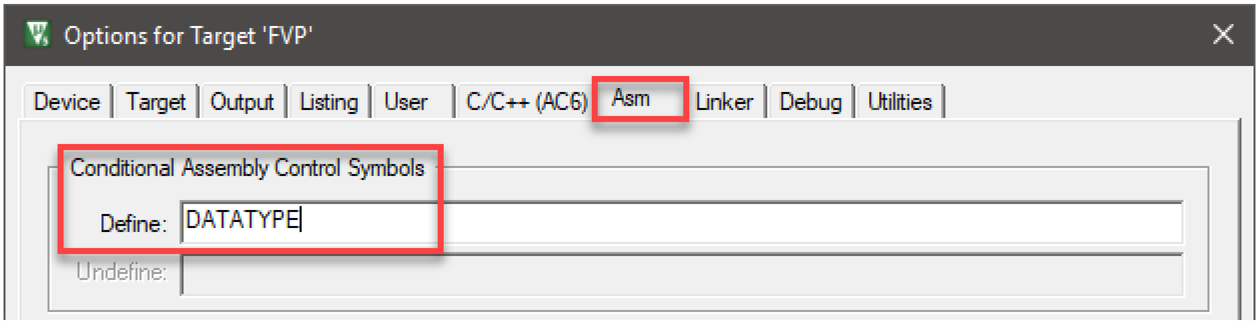
Q0	uint32	□ Hex
Q1	uint32	☑ Hex
Q2	uint32	□ Hex
Q3	uint32	□ Hex
Q4	uint32	□ Hex
Q5	uint32	□ Hex
Q6	uint32	□ Hex
Q7	uint32	□ Hex

The **Vectors** area displays the values of vectors Q0 - Q7. The Cortex-M55 works in parallel on 2 x 64-bit vectors. You can configure the display of this window to show the native number format that you are using in your algorithms, from 64-bit down to 8-bit. You can specify to see the content of the vector register in int, float, or even q number format. This makes it easy to verify the correct operation of your application.

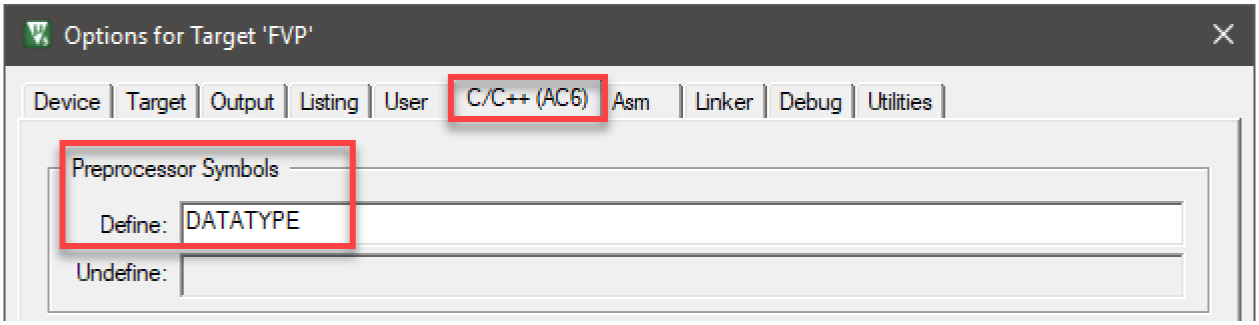
FVP – Data type optimized vector implementation

To see the best performance of the MLA algorithm, we need not only to use an optimized variant, but also change the data type. When creating DSP applications, developers often work on 16-bit values and fixed-point representation. This data type usually provides enough precision but offers a smaller memory footprint and increases performance. Combined with MVE, this means up to eight different elements can be processed in a single iteration of a loop.

To examine the impact on performance of switching to use a smaller data type, define `DATATYPE` in the **Options for Target – Asm** (Atl+F7) **Define:** section and **Options for Target – C/C++ (AC6) Define:** section:

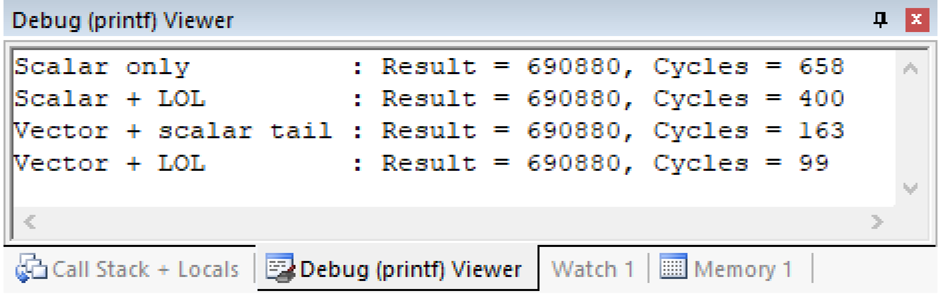


Define: DATATYPE



Define: DATATYPE

Rebuild the project, **Start a Debug Session** (Ctrl + F5), and **Run** (F5) the application. You should see results like the following ones:



```
Scalar only          : Result = 690880, Cycles = 658
Scalar + LOL         : Result = 690880, Cycles = 400
Vector + scalar tail : Result = 690880, Cycles = 163
Vector + LOL         : Result = 690880, Cycles = 99
```

Comparing the different results, we see that an optimized implementation of an algorithm with the right selection of the variable data types can increase performance significantly. In our case, the Vector + LOL version is more than 6.5 times faster than the simple scalar implementation. Let's see how the numbers are on a real hardware implementation in an FPGA.

MPS3 - Data type optimized vector implementation

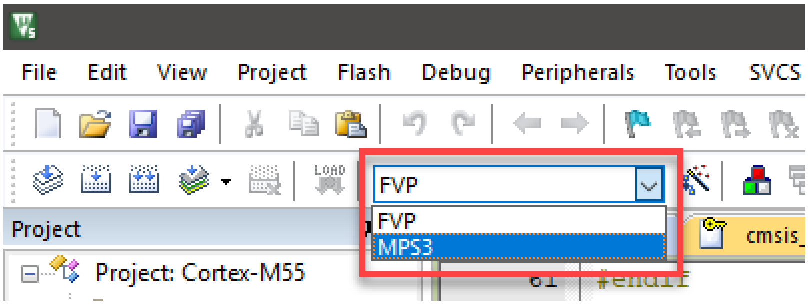
For this next part, you need to have access to the Arm MPS3 FPGA Prototyping Board. It features a large FPGA that lets designers implement complex embedded designs. It offers various debug connectors, including the 20-pin Cortex + ETM debug connector. Connecting a ULINKpro debug and trace unit to this connector gives access to full instruction trace that you can use for code coverage and profiling.

Hardware setup

1. Connect the ULINKpro to your computer and to J12 on the MPS3
2. Connect the MPS3 power supply
3. Press the PBON button to load the FPGA image

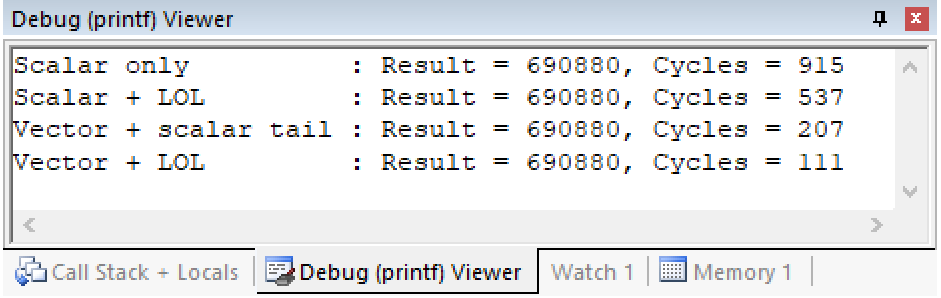
Run the application

In µVision, switch the target to **MPS3**:



Make sure that the define DATATYPE is still set on the **C/C++ (AC6)** and **Asm** tabs.

Build (F7) the project, **Start a Debug Session** (Ctrl + F5), and **Run** (F5) the application. It will hit a breakpoint at the end on the `while(1)` loop. Observe the output of the `printf()` calls in the **Debug (printf) Viewer** window:

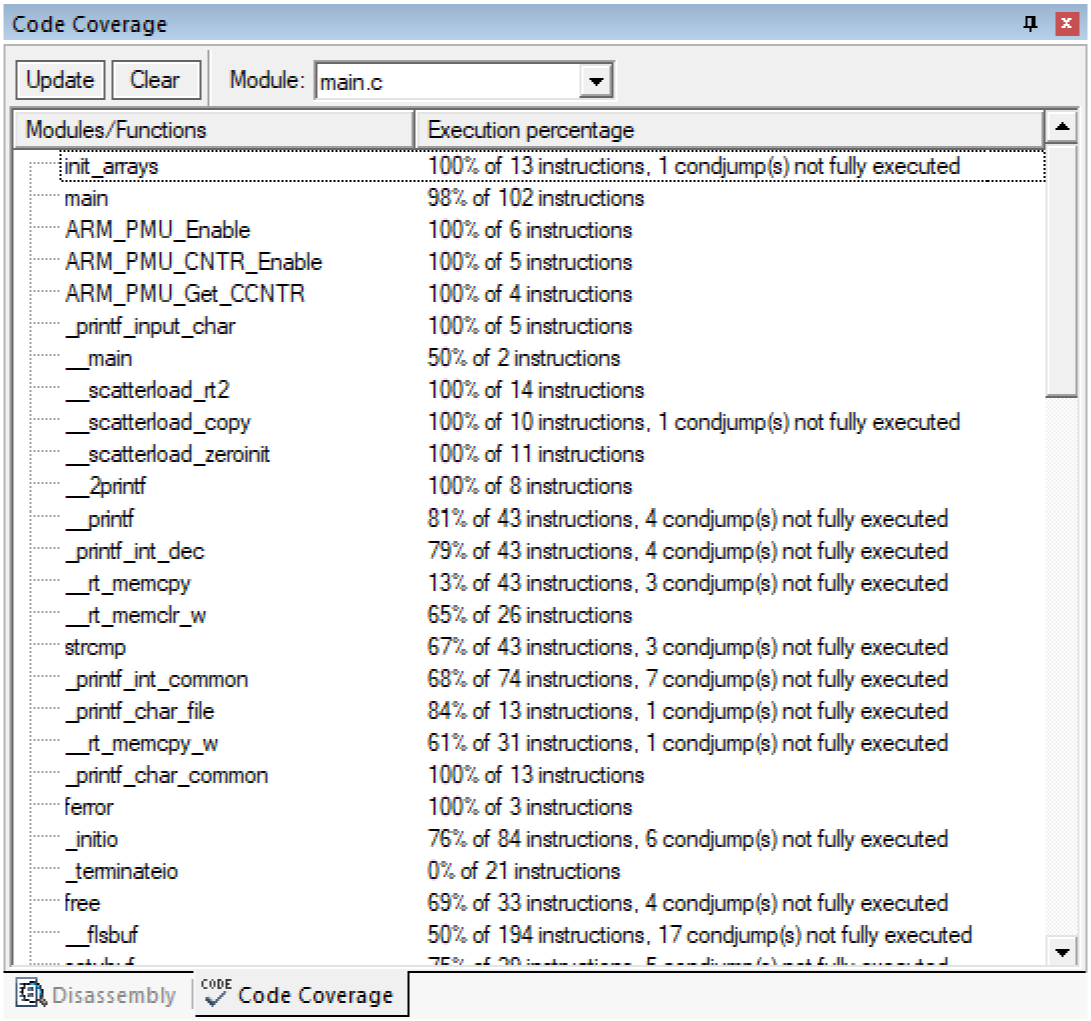


```
Scalar only          : Result = 690880, Cycles = 915
Scalar + LOL         : Result = 690880, Cycles = 537
Vector + scalar tail : Result = 690880, Cycles = 207
Vector + LOL         : Result = 690880, Cycles = 111
```

Notice that the Vector + LOL implementation needs more cycles than estimated with the model, but the relation to the Scalar implementation is roughly right: in real life it is even more than eight times faster than the simple implementation.

Code Coverage

Notice that the code is already annotated with coverage information once you run through it. This is an advantage when using real hardware. You can also use the **Code Coverage** window to check the coverage for each module/function:



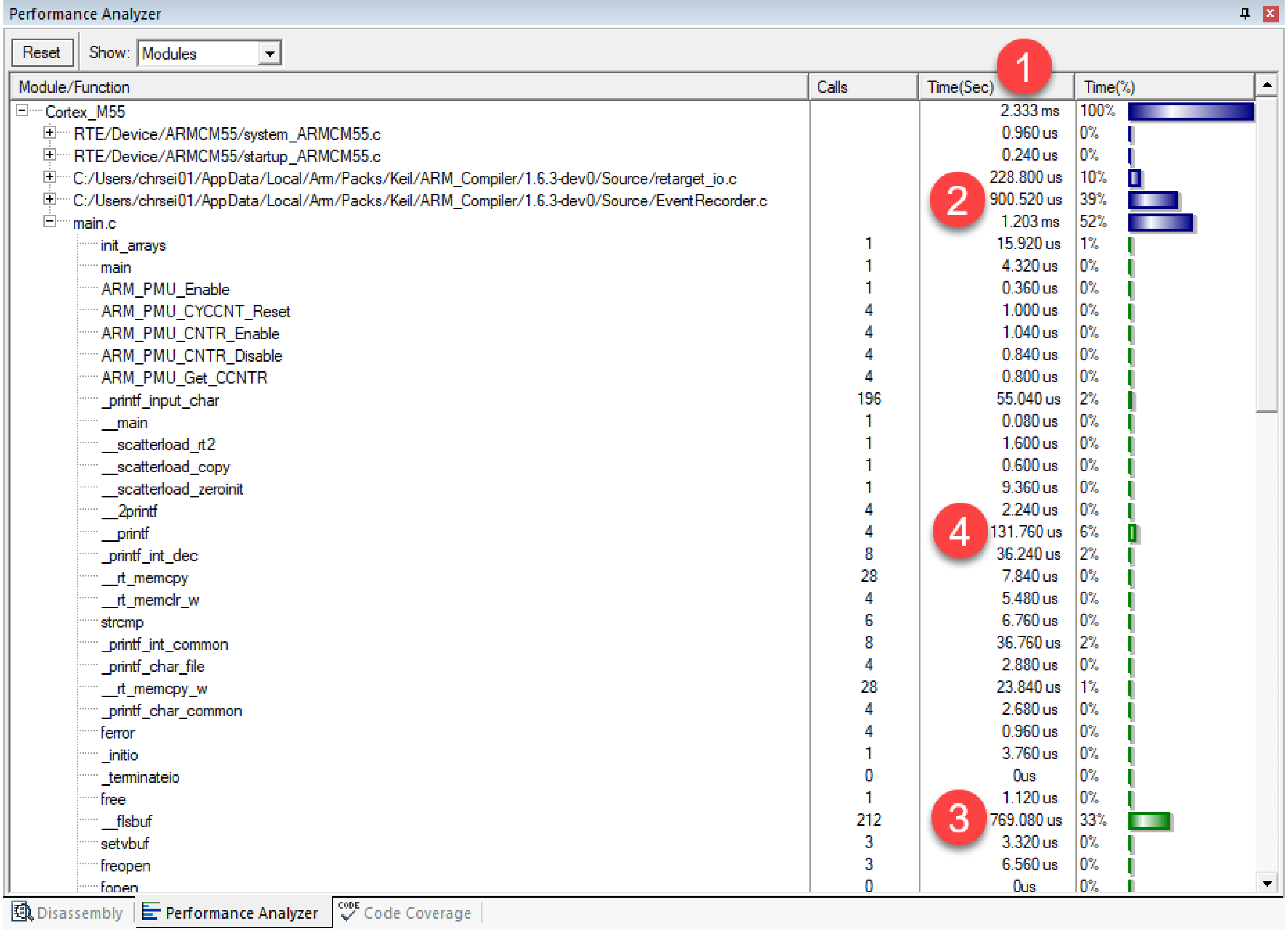
Modules/Functions	Execution percentage
init_arrays	100% of 13 instructions, 1 condjump(s) not fully executed
main	98% of 102 instructions
ARM_PMU_Enable	100% of 6 instructions
ARM_PMU_CNTR_Enable	100% of 5 instructions
ARM_PMU_Get_CCNTR	100% of 4 instructions
_printf_input_char	100% of 5 instructions
__main	50% of 2 instructions
__scatterload_rt2	100% of 14 instructions
__scatterload_copy	100% of 10 instructions, 1 condjump(s) not fully executed
__scatterload_zeroinit	100% of 11 instructions
__2printf	100% of 8 instructions
__printf	81% of 43 instructions, 4 condjump(s) not fully executed
_printf_int_dec	79% of 43 instructions, 4 condjump(s) not fully executed
__rt_memcpy	13% of 43 instructions, 3 condjump(s) not fully executed
__rt_memclr_w	65% of 26 instructions
strcmp	67% of 43 instructions, 3 condjump(s) not fully executed
_printf_int_common	68% of 74 instructions, 7 condjump(s) not fully executed
_printf_char_file	84% of 13 instructions, 1 condjump(s) not fully executed
__rt_memcpy_w	61% of 31 instructions, 1 condjump(s) not fully executed
_printf_char_common	100% of 13 instructions
ferror	100% of 3 instructions
_initio	76% of 84 instructions, 6 condjump(s) not fully executed
_terminateio	0% of 21 instructions
free	69% of 33 instructions, 4 condjump(s) not fully executed
__flsbuf	50% of 194 instructions, 17 condjump(s) not fully executed
_setvbuf	75% of 20 instructions, 5 condjump(s) not fully executed

As before, you can write coverage information to a Gcov file for further processing.

Performance Analyzer

Using ETM trace you also get access to **Performance Analyzer**. ULINK*pro* allows applications to be run for long periods of time while collecting trace information. This is used by **Performance Analyzer** to record and display execution times for functions and program blocks. It shows the processor cycle usage and enables you to identify algorithms that require optimization.

Go to **View – Analysis Windows – Performance Analyzer** to open the window:



Module/Function	Calls	Time(Sec)	Time(%)
Cortex_M55		2.333 ms	100%
RTE/Device/ARMCM55/system_ARMCM55.c		0.960 us	0%
RTE/Device/ARMCM55/startup_ARMCM55.c		0.240 us	0%
C:/Users/chrsei01/AppData/Local/Arm/Packs/Keil/ARM_Compiler/1.6.3-dev0/Source/retarget_io.c		228.800 us	10%
C:/Users/chrsei01/AppData/Local/Arm/Packs/Keil/ARM_Compiler/1.6.3-dev0/Source/EventRecorder.c		900.520 us	39%
main.c		1.203 ms	52%
init_arrays	1	15.920 us	1%
main	1	4.320 us	0%
ARM_PMU_Enable	1	0.360 us	0%
ARM_PMU_CYCCNT_Reset	4	1.000 us	0%
ARM_PMU_CNTR_Enable	4	1.040 us	0%
ARM_PMU_CNTR_Disable	4	0.840 us	0%
ARM_PMU_Get_CCNTR	4	0.800 us	0%
_printf_input_char	196	55.040 us	2%
__main	1	0.080 us	0%
__scatterload_rt2	1	1.600 us	0%
__scatterload_copy	1	0.600 us	0%
__scatterload_zeroinit	1	9.360 us	0%
__2printf	4	2.240 us	0%
__printf	4	131.760 us	6%
_printf_int_dec	8	36.240 us	2%
__rt_memcpy	28	7.840 us	0%
__rt_memclr_w	4	5.480 us	0%
strcmp	6	6.760 us	0%
_printf_int_common	8	36.760 us	2%
_printf_char_file	4	2.880 us	0%
__rt_memcpy_w	28	23.840 us	1%
_printf_char_common	4	2.680 us	0%
ferror	4	0.960 us	0%
_initio	1	3.760 us	0%
_terminateio	0	0us	0%
free	1	1.120 us	0%
__flsbuf	212	769.080 us	33%
setvbuf	3	3.320 us	0%
freopen	3	6.560 us	0%
fopen	0	0us	0%

The following is quite interesting:

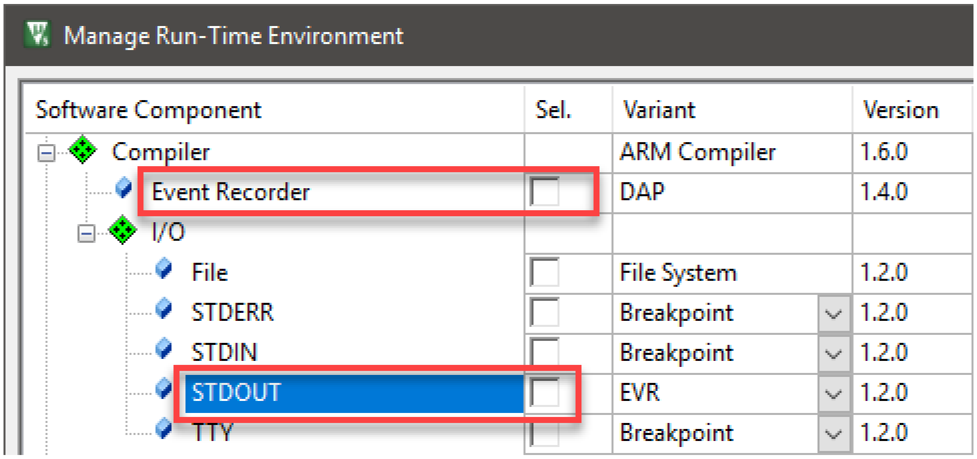
1. The overall execution time is 2.333 ms
2. Almost 40% of the execution time was spent in the Event Recorder component
3. `__flsbuf` (the Arm implementation of `fputc`) is eating up another 33%
4. `printf` itself requires another 6%

Putting all together, we can see that in such a simple project, the dominating factor is printing the output to a console. In the next section, we will see how we can improve this.

MPS3 – Reducing Execution Time Using Component Viewer

Stop the Debug Session (Ctrl + F5) and open the **Manage Run-Time Environment** window. Disable the following components:

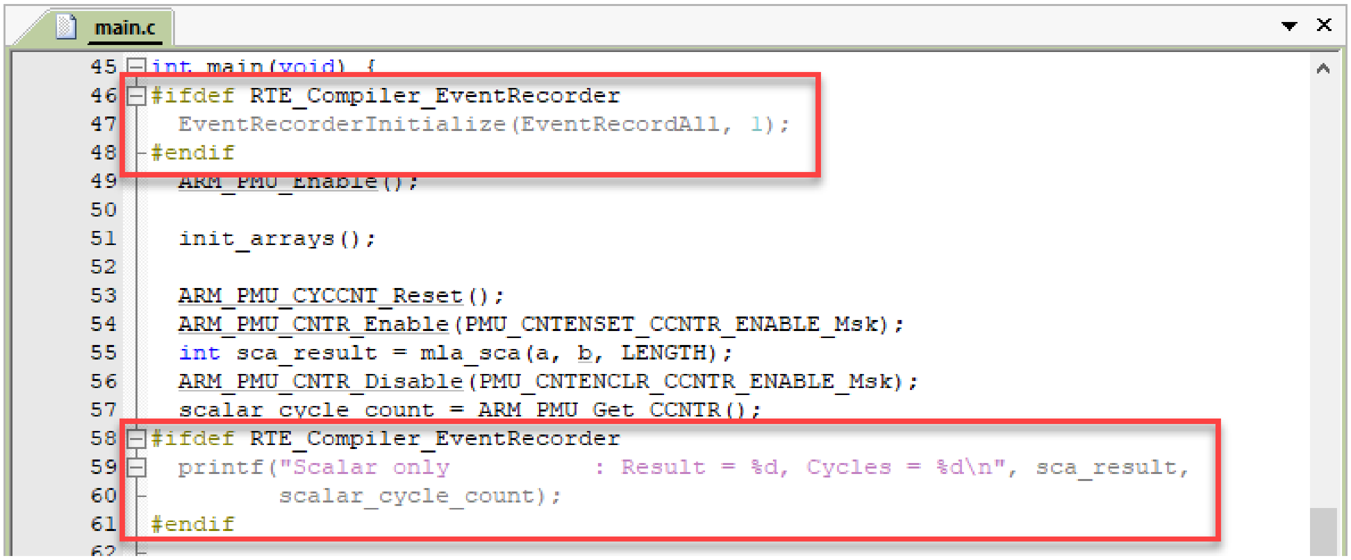
- **::Compiler:Event Recorder**
- **::Compiler:I/O:STDOUT**



Software Component	Sel.	Variant	Version
Compiler		ARM Compiler	1.6.0
Event Recorder	□	DAP	1.4.0
I/O			
File	□	File System	1.2.0
STDERR	□	Breakpoint	1.2.0
STDIN	□	Breakpoint	1.2.0
STDOUT	□	EVR	1.2.0
TTY	□	Breakpoint	1.2.0

Click **OK** to close the window.

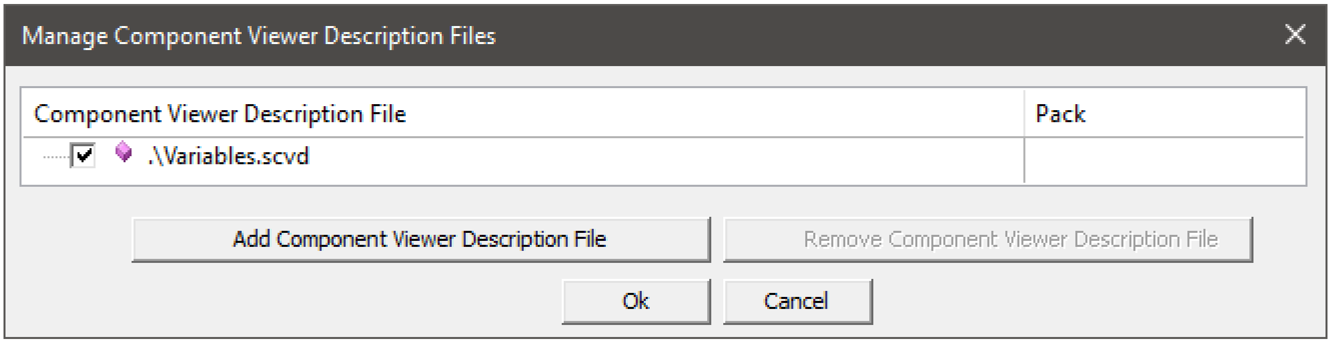
In `main.c`, you immediately see that the relevant code will not be used as the `RTE_Component.h` file does not contain the define `RTE_Compiler_EventRecorder` anymore:



```
int main(void) {
#ifdef RTE_Compiler_EventRecorder
  EventRecorderInitialize(EventRecordAll, 1);
#endif
  ARM_PMU_Enable();

  init_arrays();

  ARM_PMU_CYCCNT_Reset();
  ARM_PMU_CNTR_Enable(PMU_CNTENSET_CCNTR_ENABLE_Msk);
  int sca_result = mla_sca(a, b, LENGTH);
  ARM_PMU_CNTR_Disable(PMU_CNTENCLR_CCNTR_ENABLE_Msk);
  scalar_cycle_count = ARM_PMU_Get_CCNTR();
#ifdef RTE_Compiler_EventRecorder
  printf("Scalar only          : Result = %d, Cycles = %d\n", sca_result,
         scalar_cycle_count);
#endif
```

Instead of using `printf()` for displaying the variables, you could add them to the **Watch** window, but this requires the variables to be in scope and does not render nicely (the variable name and value is shown, that's it). Another option is to use **Component Viewer** to create your own window to view to the variables by just adding a simple XML file to the project.

Go to **Options for Target – Debug** (Atl+F7) and click **Manage Component Viewer Description Files …** at the bottom of the dialog. In the next window, click **Add Component Viewer Description File** and browse to the file `Variables.scvd` in the project directory, and click **Add**:


Manage Component Viewer Description Files
Component Viewer Description File
Pack
☑ .\Variables.scvd
Add Component Viewer Description File
Remove Component Viewer Description File
Ok
Cancel

Click **OK** twice.

Variables.scvd

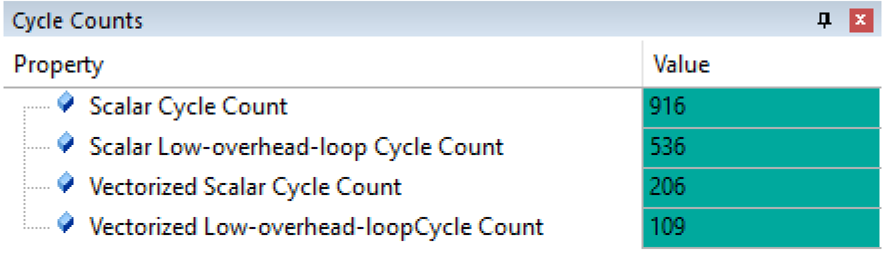
Here is the content of the SCVD file:

```
<?xml version="1.0" encoding="utf-8"?>

<component_viewer schemaVersion="0.1" xmlns:xs="http://www.w3.org/2001/XMLSchema-
instance" xs:noNamespaceSchemaLocation="Component_Viewer.xsd">

<component name="MyVars" version="1.0.0"/>
  <objects>
    <object name="MyVars">
      <read name="SCNT"  type="uint32_t" symbol="scalar_cycle_count"/>
      <read name="SLCNT" type="uint32_t" symbol="scalar_lol_cycle_count"/>
      <read name="VSCNT" type="uint32_t" symbol="vector_scalar_cycle_count"/>
      <read name="VLCNT" type="uint32_t" symbol="vector_lol_cycle_count"/>
      <out name="Cycle Counts">
        <item property="Scalar Cycle Count"                      value="%d[SCNT]" />
        <item property="Scalar Low-overhead-loop Cycle Count"    value="%d[SLCNT]"/>
        <item property="Vectorized Scalar Cycle Count"           value="%d[VSCNT]"/>
        <item property="Vectorized Low-overhead-loopCycle Count" value="%d[VLCNT]"/>
      </out>
    </object>
  </objects>

</component_viewer>
```

Basically, you create the objects, that you want to display in the window by reading program symbols. Then you tell the window how to display these objects (items) in a consistent way (using printf-style formatting).

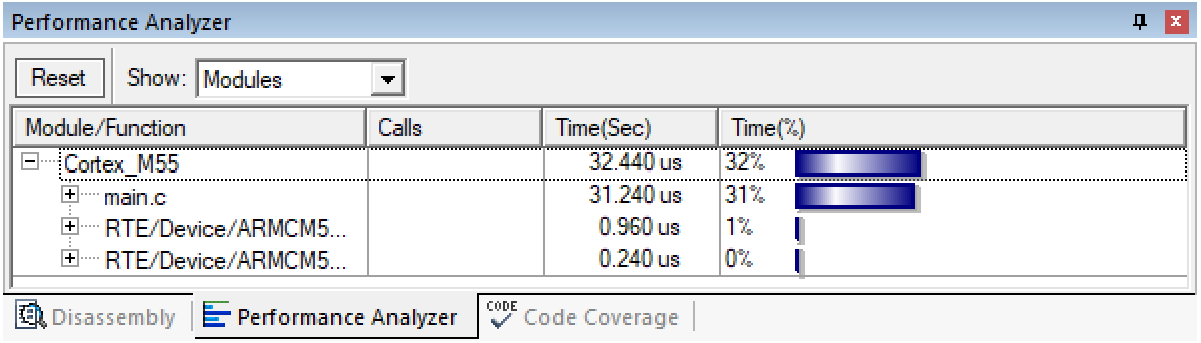
Results

Rebuild the project, **Start a Debug Session** (Ctrl + F5) and go to **View – Watch Windows – Cycle Counts** to open the Component Viewer window. **Run** (F5) the application. The **Cycle Counts** window shows the following results:



Property	Value
Scalar Cycle Count	916
Scalar Low-overhead-loop Cycle Count	536
Vectorized Scalar Cycle Count	206
Vectorized Low-overhead-loopCycle Count	109

Performance Analyzer now shows a different picture:



Show: Modules

Module/Function	Calls	Time(Sec)	Time(%)
Cortex_M55		32.440 us	32%
main.c		31.240 us	31%
RTE/Device/ARMCM5...		0.960 us	1%
RTE/Device/ARMCM5...		0.240 us	0%

Using a less invasive method of reading the variables, we could reduce the overall run-time of the application drastically.

Summary

This application note showed how you can use the Fixed Virtual Platforms (FVPs) that are shipped with Arm Keil MDK to start early prototyping of your software or architecture exploration without real hardware. It explained that this method is suitable for benchmarking your application code, while not having access to a target device.

It was also shown that using a prototyping board, advanced features of Arm Keil MDK help you to profile your application further and how these features can help to reduce the overall run-time of the program.

Appendix

Here are some useful resources regarding the Armv8.1-M architecture:

1. White paper: Introducing the new Armv8.1-M architecture
2. Armv8-M Architecture Reference Manual Documentation
3. Cortex-M55 on developer.arm.com