

# Appendix E: Dynamic Instruction Count Profiling for Cortex-M0 using IAR Embedded Workbench IDE for ARM

Dr. Ehsan Ali  
Assumption University of Thailand  
ehsanali@au.edu  
ehssan.aali@gmail.com

Nov. 2024

# Contents

<b>1</b>	<b>Appendix E: Dynamic Instruction Count Profiling for Cortex-M0 using IAR Embedded Workbench IDE for ARM</b>	<b>2</b>
1.1	System Requirement . . . . .	2
1.2	Target Specification . . . . .	2
1.3	Project Creation . . . . .	2
1.3.1	Processor Selection . . . . .	3
1.3.2	Linker Script . . . . .	3
1.3.3	ARM Simulator . . . . .	7
1.3.4	IAR Embedded Workbench - Memory Configuration . . . . .	7
1.4	Instruction Count Profiling . . . . .	8

# Chapter 1

## Appendix E: Dynamic Instruction Count Profiling for Cortex-M0 using IAR Embedded Workbench IDE for ARM

### 1.1 System Requirement

This is Appendix E for article "Innovative Hardware Accelerator Architecture for FPGA-Based General-Purpose RISC Microprocessors".

1. x86 or x86\_64 host machine running Windows operating system.
2. Licensed IAR Embedded Workbench IDE for ARM version 8.40.1.21539.

### 1.2 Target Specification

- Core: ARMv6m Cortex-M0
- ROM1: Capacity: 1.255 MB, memory address origin: 0x0
- ROM2: Capacity: 128 KB, memory address origin: 0x120000
- RAM1: Capacity: 4MB, memory address origin: 0x80000
- Initial stack pointer: 0xF400
- Initial procedure pointer: 0x0 (Not used)
- Initial heap pointer: 0xF800

### 1.3 Project Creation

Either create a C/C++ program and enter your algorithm in C/C++ programming languages and then compile it using IAR compiler or enter the C/C++ code in your favorite editor and then compile it according to the guide in Appendix B using LLVM clang compiler and then import the generated ELF file into IAR by selecting "Create New Project" - "Externally built executable".

### 1.3.1 Processor Selection

Right click on project name → "Options.." → "General Options" and set the Processor variant to Core=Cortex-M0 as shown in Fig. 1.1.

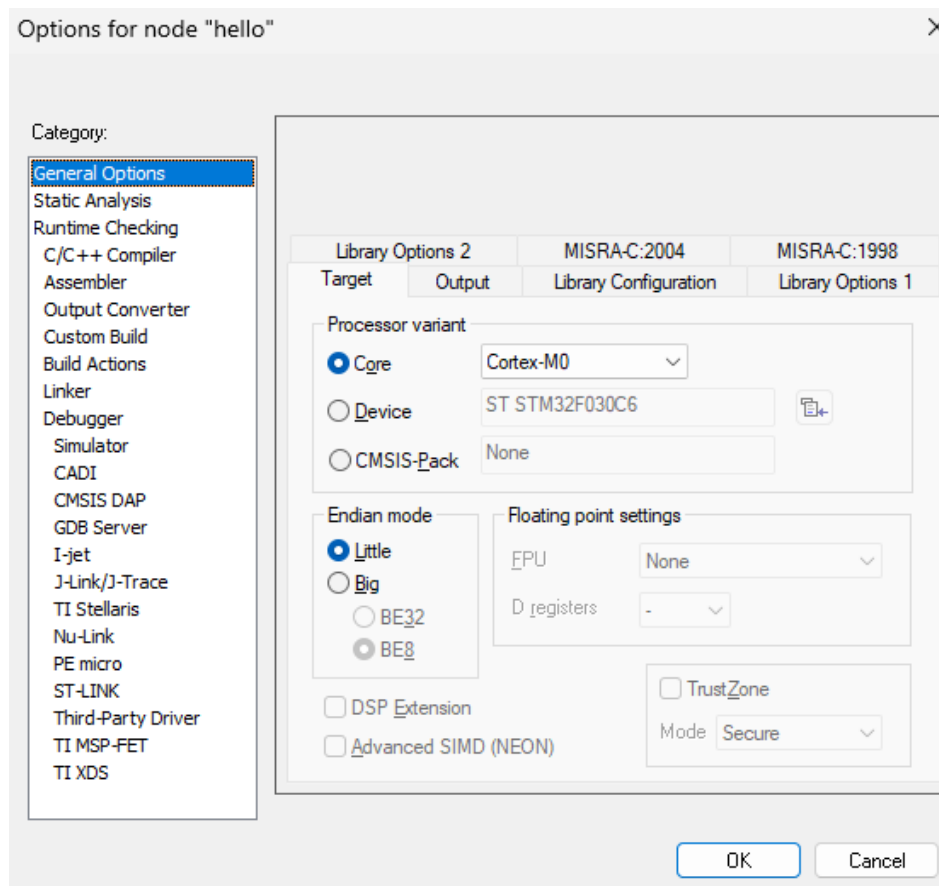


Figure 1.1: IAR Embedded Workbench - Project Linker Options - General Options.

### 1.3.2 Linker Script

The following linker script must be pass to the LLVM clang compiler using -T parameter. Change the memory configuration according to your hardware.

Listing 1.1: "cortex-m0.ld"

```
1 ENTRY( main )
2 MEMORY {
3     ROM(rx)      : ORIGIN = 0x0,          LENGTH = 0x11FFFF          /* 1.125 MB */
4     ROM2(rx)     : ORIGIN = 0x120000,     LENGTH = 0x20000          /* 128 KB */
5     RAM(rwx)     : ORIGIN = 0x800000,     LENGTH = 0x400000          /* 4 MB */
6 }
7
8 SECTIONS {
9     /* The code should be loaded at address 0x0 */
10    /* the dot (.) symbol is location counter */
11    . = ORIGIN(ROM);
```

```

12  .text : {
13      /* When link-time garbage collection is in use ('--gc-sections'),
14      it is often useful to mark sections that should not be eliminated.
15      This is accomplished by surrounding an input section's wildcard
16      entry with KEEP() */
17      KEEP(*(.vector_table));
18      . = ALIGN(4);
19      __vec_end__ = .;
20      /* Input sections: .text section in all files. */
21      * (.text)
22      . = ALIGN(4);
23      __end_text__ = .;
24  } > ROM /* assign .text section to a previously defined region of
25          memory 'ROM' */
26
27  . = ORIGIN(ROM2);
28  .rodata : {
29      __dataro_start__ = .;
30      * (.rodata)
31      . = ALIGN(4);
32      __dataro_end__ = .;
33  } > ROM2
34
35  . = ORIGIN(RAM);
36  .data : {
37      __data_start__ = .;
38      * (.data);
39      . = ALIGN(4);
40      __heap_low = .;      /* for _sbrk */
41      . = . + 0x10000;      /* 64kB of heap memory */
42      __heap_top = .;      /* for _sbrk */
43      __data_end__ = .;
44  } > RAM

```

Using the values stated in our linker script (Listing 1.1) the linker options for IAR Workbench IDE project must be set accordingly.

Right click on project name → "Options.." → Linker and the tick on "Override default" as shown in Fig. 1.2.

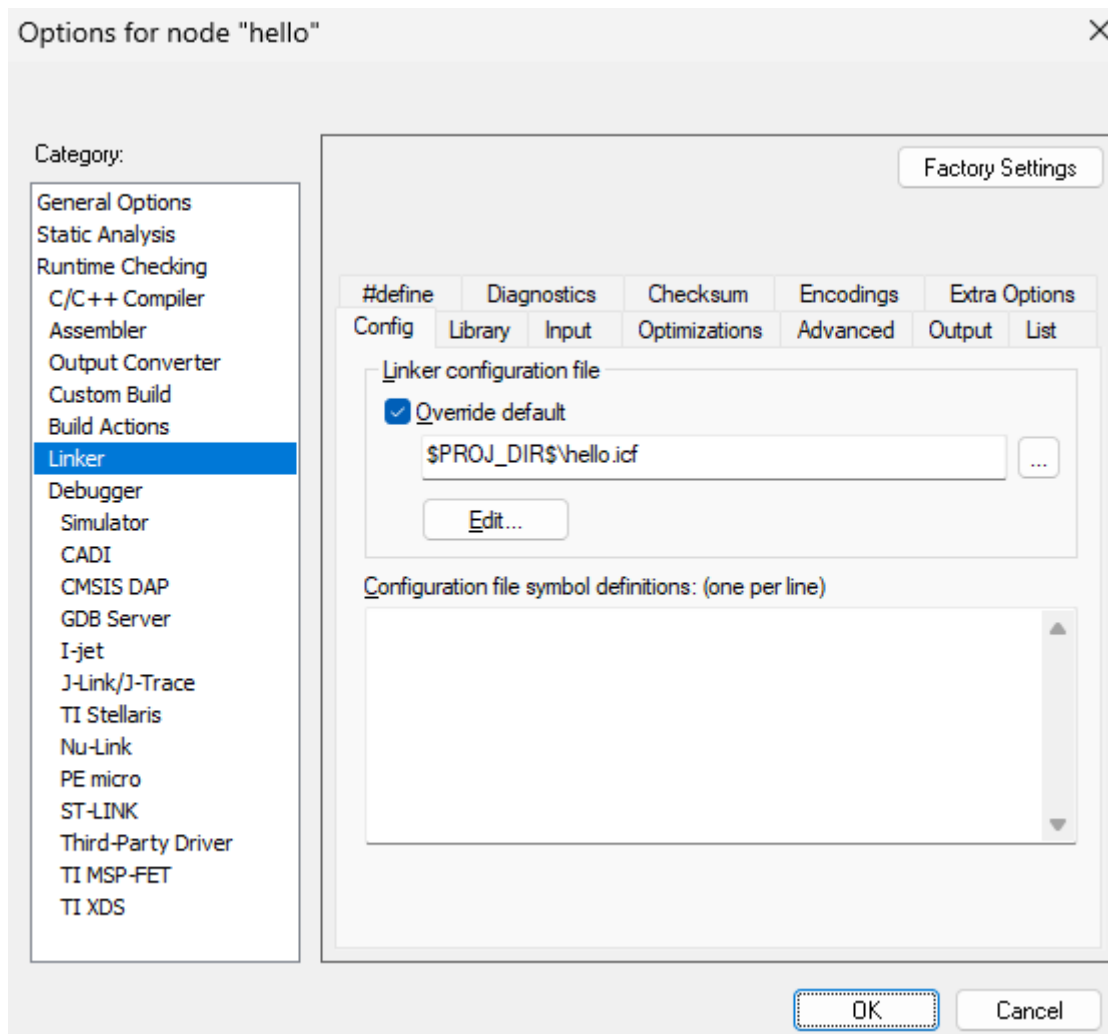


Figure 1.2: IAR Embedded Workbench - Project Linker Options.

Click the "Edit" button and define the memory regions according to the hardware value in the original linker script (Listing 1.1 as shown in Fig. 1.3).

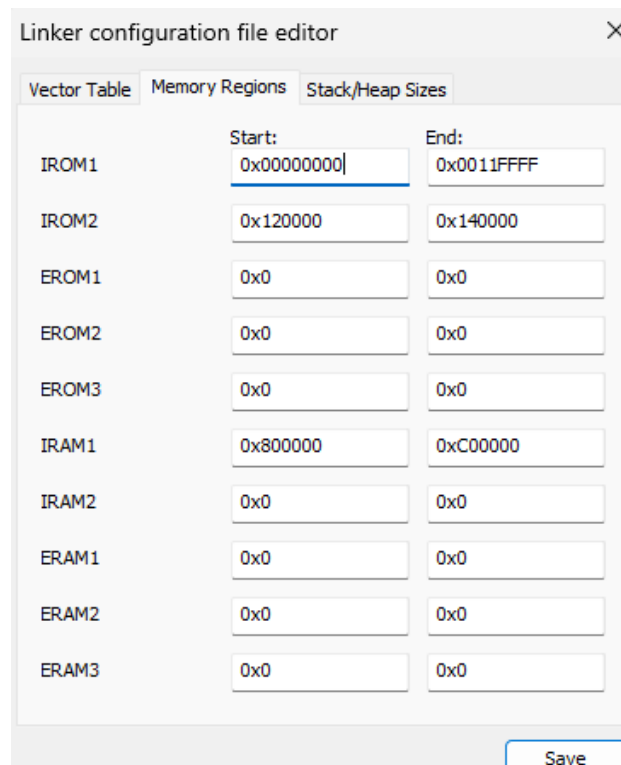


Figure 1.3: IAR Embedded Workbench - Project Linker Options - Memory Regions.

Set the size of stack and heap memory as shown in Fig. 1.4.

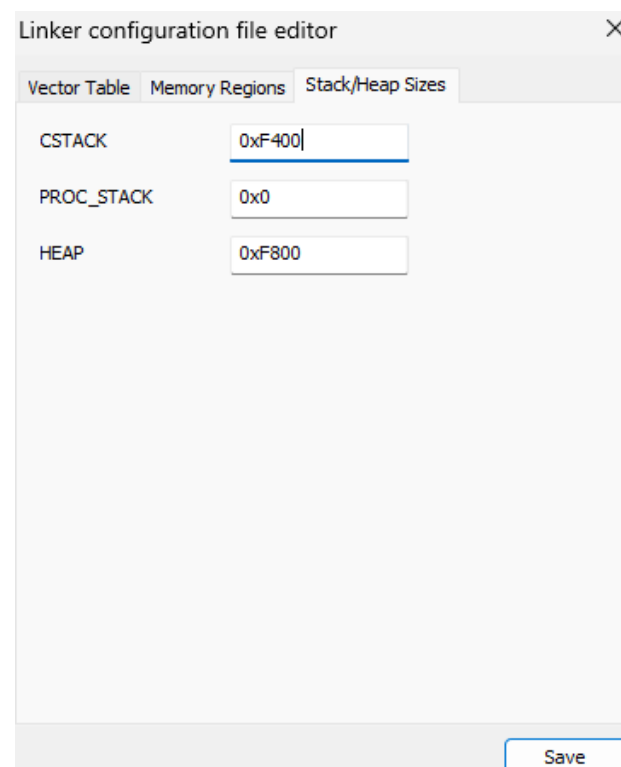


Figure 1.4: IAR Embedded Workbench - Project Linker Options - Stack/Heap Sizes.

### 1.3.3 ARM Simulator

Right click on project name → "Options.." → "Debugger" and set the Driver to Simulator as shown in Fig. 1.5.

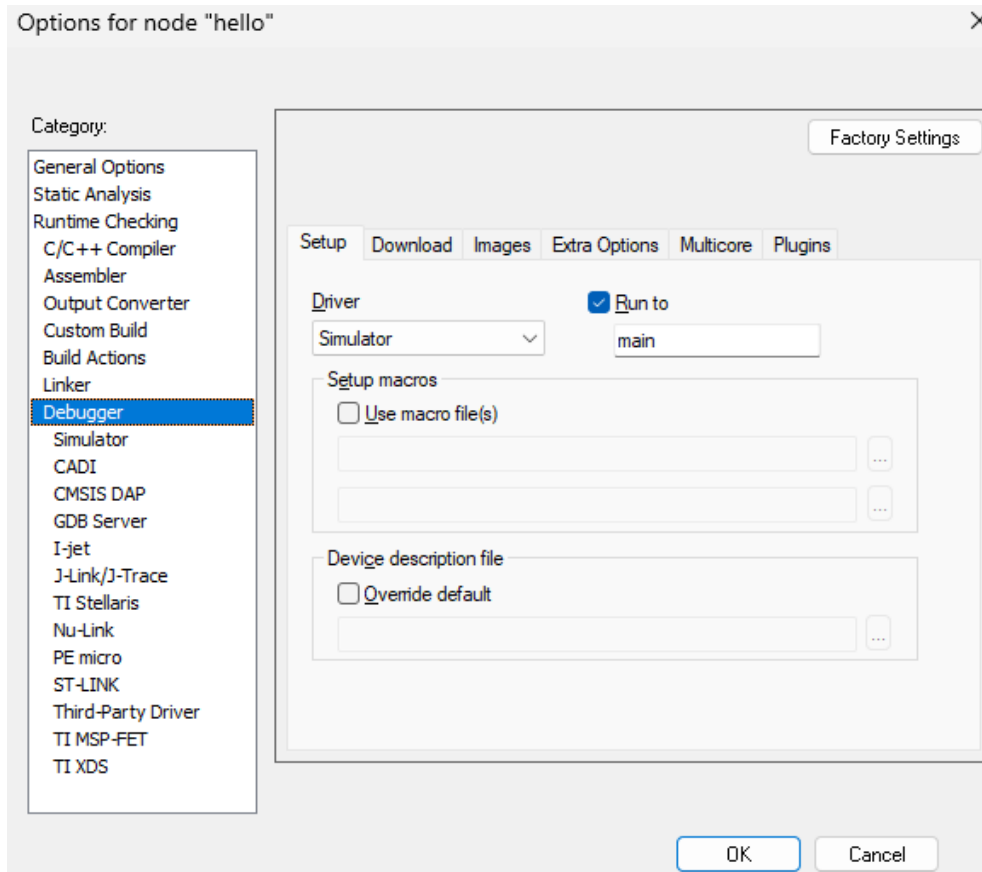


Figure 1.5: IAR Embedded Workbench Project - Debugger options.

### 1.3.4 IAR Embedded Workbench - Memory Configuration

Set the memory configuration by stating the start, end, and type of memory according to the hardware and values in the linker script (Listing 1.1) as shown in Fig. 1.6.



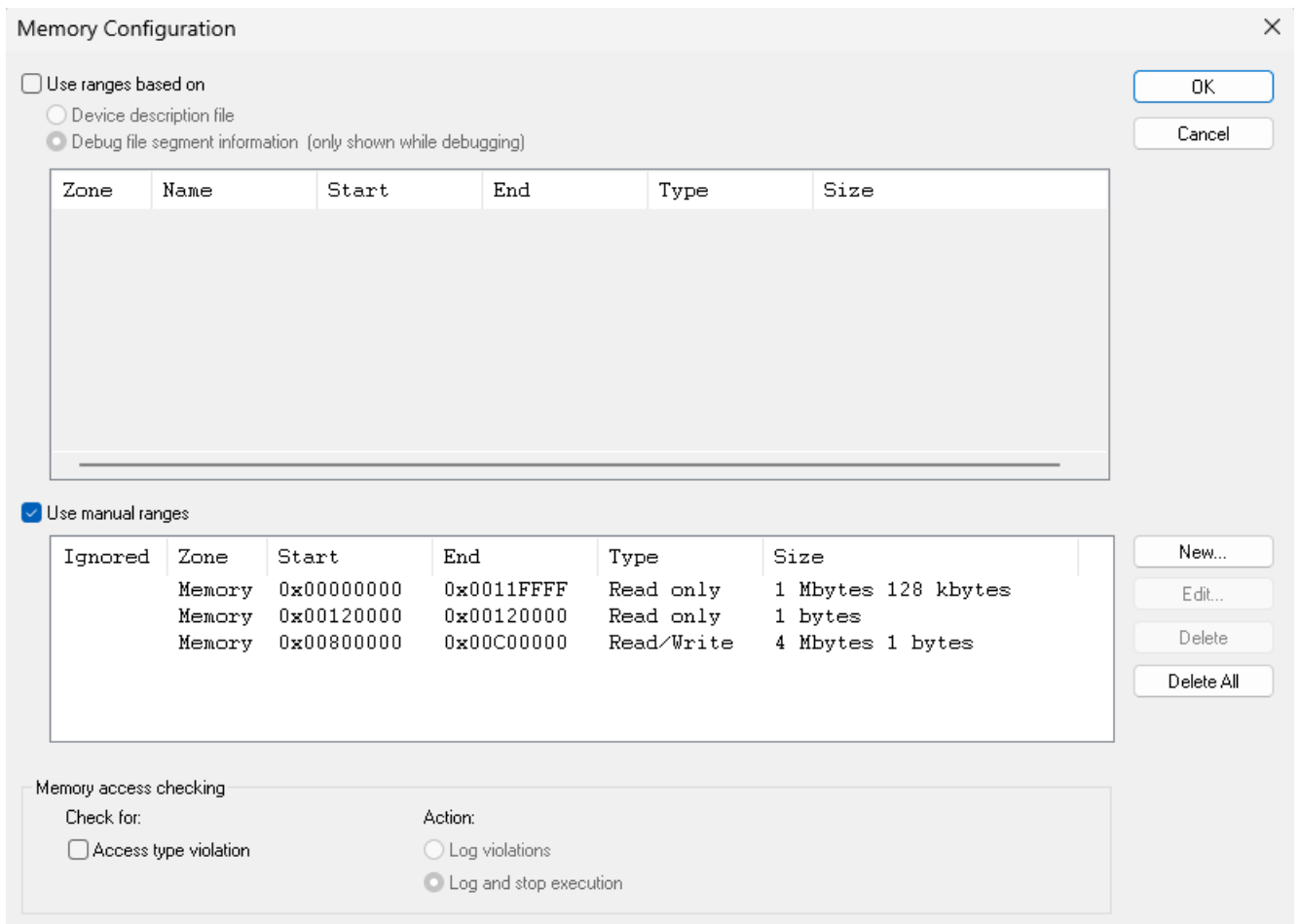


Figure 1.6: IAR Embedded Workbench Project - Memory Configuration

## 1.4 Instruction Count Profiling

First build the project: "Project" → "Rebuild All".

Set a breakpoint on return line of the `main()` function and then issue the "Download and Debug" command (Ctrl+D).

Fig. 1.7 shows several profiling and trace options alongside the register bank content instruction set disassemble windows. The screenshot shows the simulation of FFT algorithm by calling the `fft()` function from `main()`.

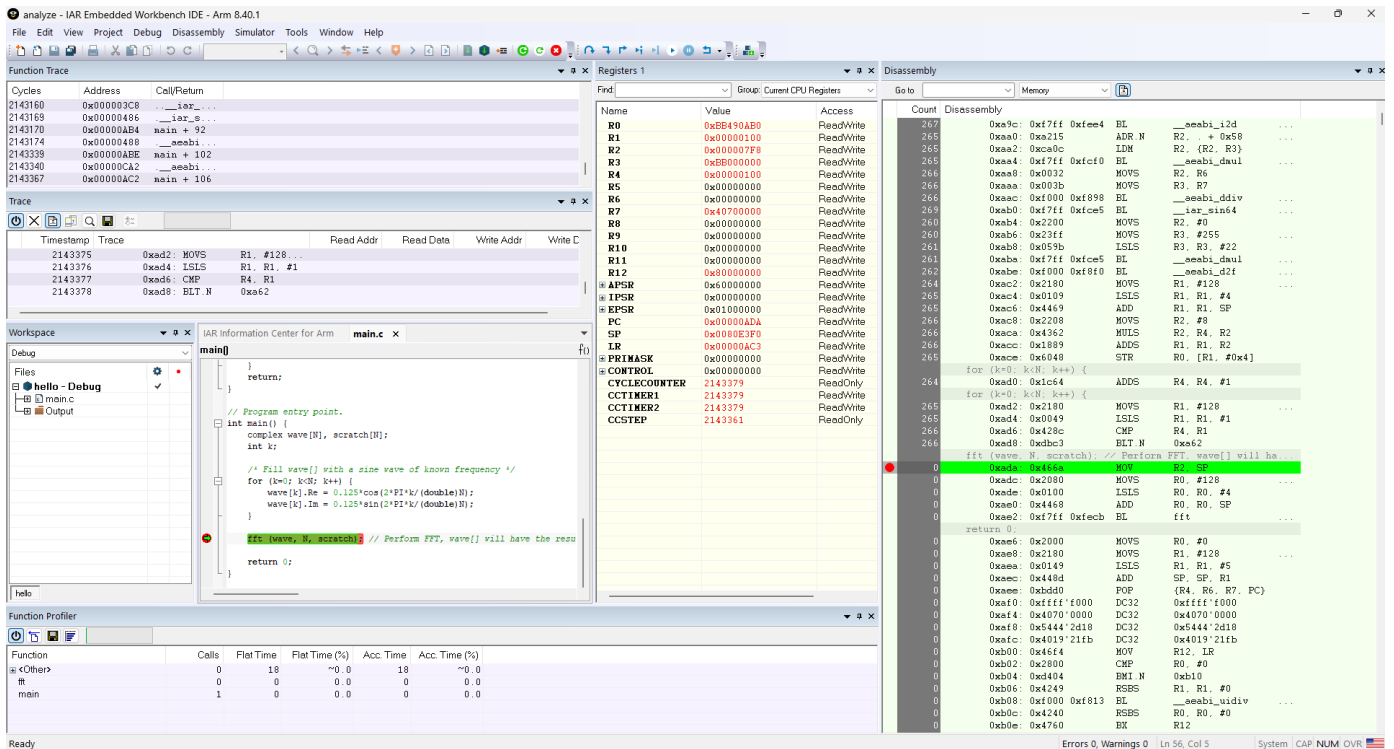


Figure 1.7: IAR Embedded Workbench - Debugging with Simulator Driver.

Fig. 1.8 shows the disassembly where the first column states the number of times the disassembled instruction is executed. It can be easily seen that code block residing at memory location 0x3c2-0x3de (marked with blue rectangle) runs twice more than the code residing at 0x370-0x3c0 while instructions at 0x3e0 onward is never executed.

Notice the frequent occurrence of [MOVS, BL] instruction pair (marked with red rectangles).

A context menu with "Copy Window Content" item is provided which can be access by right clicking on the disassembly window. You can copy the profiling result into a text file and use a scripting language to automatically extract the code blocks with highest instruction count (blue rectangle) or identify the most frequent instruction pairs (red rectangles).

This result is sent to the hardware accelerator code generator to produce VHDL modules as described in "Innovative Hardware Accelerator Architecture for FPGA-Based General-Purpose RISC Microprocessors" article.

Disassembly				
Go to		Memory		
Count	Disassembly			
269	0x370: 0xa234	ADR.N	R2, . + 0xd4	
268	0x372: 0xca0c	LDM	R2, {R2, R3}	
268	0x374: 0xf000 0xf963	BL	__aeabi_dadd	
269	0x378: 0x0032	MOVS	R2, R6	
268	0x37a: 0x003b	MOVS	R3, R7	
268	0x37c: 0xf000 0xf884	BL	__aeabi_dmul	
268	0x380: 0xa232	ADR.N	R2, . + 0xcc	
267	0x382: 0xca0c	LDM	R2, {R2, R3}	
268	0x384: 0xf000 0xf95b	BL	__aeabi_dadd	
263	0x388: 0x0032	MOVS	R2, R6	
264	0x38a: 0x003b	MOVS	R3, R7	
264	0x38c: 0xf000 0xf87c	BL	__aeabi_dmul	
269	0x390: 0xa230	ADR.N	R2, . + 0xc4	
269	0x392: 0xca0c	LDM	R2, {R2, R3}	
270	0x394: 0xf000 0xf953	BL	__aeabi_dadd	
267	0x398: 0x0032	MOVS	R2, R6	
267	0x39a: 0x003b	MOVS	R3, R7	
267	0x39c: 0xf000 0xf874	BL	__aeabi_dmul	
263	0x3a0: 0xa22e	ADR.N	R2, . + 0xbc	
262	0x3a2: 0xca0c	LDM	R2, {R2, R3}	
261	0x3a4: 0xf000 0xf94b	BL	__aeabi_dadd	
267	0x3a8: 0x0032	MOVS	R2, R6	
266	0x3aa: 0x003b	MOVS	R3, R7	
265	0x3ac: 0xf000 0xf86c	BL	__aeabi_dmul	
263	0x3b0: 0xa22c	ADR.N	R2, . + 0xb4	
264	0x3b2: 0xca0c	LDM	R2, {R2, R3}	
264	0x3b4: 0xf000 0xf943	BL	__aeabi_dadd	
260	0x3b8: 0x9a00	LDR	R2, [SP]	
260	0x3ba: 0x9b01	LDR	R3, [SP, #0x4]	
260	0x3bc: 0xf000 0xf864	BL	__aeabi_dmul	
260	0x3c0: 0x0022	MOVS	R2, R4	
260	0x3c2: 0x002b	MOVS	R3, R5	
525	0x3c4: 0xf000 0xf93b	BL	__aeabi_dadd	
514	0x3c8: 0x0004	MOVS	R4, R0	
514	0x3ca: 0x000d	MOVS	R5, R1	
522	0x3cc: 0x9802	LDR	R0, [SP, #0x8]	
521	0x3ce: 0x0780	LSLS	R0, R0, #30	
521	0x3d0: 0xd502	BPL.N	0x3d8	
260	0x3d2: 0x2080	MOVS	R0, #128	
260	0x3d4: 0x0600	LSLS	R0, R0, #24	
260	0x3d6: 0x4045	EORS	R5, R5, R0	
520	0x3d8: 0x0020	MOVS	R0, R4	
520	0x3da: 0x0029	MOVS	R1, R5	
520	0x3dc: 0xb003	ADD	SP, SP, #0xc	
515	0x3de: 0xbdf0	POP	{R4-R7, PC}	
0	0x3e0: 0x3cd3'145c	DC32	0x3cd3'145c	
0	0x3e4: 0x5000'0000	DC32	0x5000'0000	
0	0x3e8: 0x41b9'21fb	DC32	0x41b9'21fb	
0	0x3ec: 0x6000'0000	DC32	0x6000'0000	

Figure 1.8: IAR Embedded Workbench - Instruction Count Profiling Result.

